Required Information Release

Required Information Release

Stephen Chong

School of Engineering and Applied Sciences

Harvard University, Cambridge, MA 02138, USA

chong@seas.harvard.edu

Abstract

Many computer systems have a functional requirement to release information. Such requirements are an important part of a system's information security requirements. Current information-flow control techniques are able to reason about permitted information flows, but not required information flows.

In this paper, we introduce and explore the specification and enforcement of *required information release* in a language-based setting. We define semantic security conditions that express both *what* information a program is required to release, and *how* an observer is able to learn this information. We also consider the relationship between permitted and required information release, and define *bounded release*, which provides upper- and lower-bounds on the information a program releases. We show that both required information release and bounded release can be enforced using a security-type system.

Keywords

Information flow, declassification, information release, algorithmic knowledge.

1 Introduction

Information-flow control holds the promise of strong, end-to-end, application-specific information security [38]. To date, most research on information-flow control has focused on what flows are permitted or prohibited in a system. For example, *noninterference* [19] prohibits confidential inputs flowing to public outputs.

Many computer systems release (or *declassify*) confidential information as part of their intended functionality, and as such, violate noninterference. Much work in recent years has considered weakening noninterference to permit some flow of confidential inputs to public outputs (e.g., [26, 12, 13, 40, 43, 42, 6]).

This article is an extended version of the paper "Required Information Release," by Stephen Chong, which appears in the *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, IEEE Computer Society, 2010.

However, many systems have more than just *permission* to release information; they have an *obligation* to release information. In this work, we introduce and explore the specification and enforcement of *required information release*, or simply, *required release*.

Examples abound of systems with an obligation to release information. The Sarbanes-Oxley Act of 2002 is a United States federal law that was enacted after a series of corporate accounting scandals, and requires publicly held companies to report details of their finances to a government agency. Thus, financial systems of such companies are required to release sensitive financial information to the government agency. Pharmaceutical companies in many countries are required to report all results of clinical trials of new drugs to a government agency (such as the Food and Drug Administration) to receive approval. Computer systems that support the conduct of clinical trials must release all trial results, and not withhold negative results. In general, transparency of organizations and processes requires the release of sensitive information. Other systems that are required to release information include the following.

- *Sealed bid auctions*: at the end of the auction, the winning bid (and, depending on the auction, the winner's identity) is required to be released.
- *Information purchase*: once a customer has paid for information (such as electronic media), the information is required to be available for download.
- *Games*: legal game-play often requires release of a player's secret information, such as the cards in a player's hand, or the location of battleships on a player's board.
- *Course management system*: when a professor indicates that exam results are available, the system is required to allow students view their grades.
- *Credit card sales*: the receipt for a credit card purchase is required to show the final four digits of the customer's credit card number.

In the examples above, the required release of information is an important aspect of each system's information security. To gain assurance in the systems' correct implementation, it is necessary both to specify the required release (and other information security requirements) and to verify that the implementation satisfies the specification.

However, the specification of required release is subtle. What does it mean for a program to satisfy the required release of information? How do we know if a program is successfully and correctly releasing information? It does not suffice for the output of a program to simply depend upon, or be influenced by, the information required for release. Surprisingly, even if the output contains the information required for release, the program may not satisfy the required release of information. We use epistemic logic, and *algorithmic knowledge* [20] in particular, to guide our definition of required information release. Required information release must specify not just *what* information is to be released, but also *how* that information is to be learned by its intended recipient.

Required release is a functional requirement on a system: program output must allow an observer to learn certain information. Noninterference and most other information flow security conditions are not functional requirements. However, required

release is an information flow security condition; it describes a *mandatory* flow of information to an observer. By contrast, most existing work in information flow considers *permitted* flows of information. In terms of dependence, permitted information flow conditions restrict how the output is permitted to depend on the inputs. For example, noninterference requires that public outputs do not depend on private inputs—if a private input changes, the public output should not change. Required information release mandates that outputs *must* depend on inputs in a way that allows an observer to learn certain information.

Required release interacts with permitted information flows in more interesting ways than other functional requirements: if a system is required to release information, then the system must also be permitted to release it. Indeed, required information release and permitted information release can be combined to specify both upper and lower bounds on the information that a system releases. We do so by defining *bounded release*, a security condition that combines required release and *delimited release* [39], and thus specifies both what information a program is required to release, and what information it is allowed to release.

For some systems, bounds on information release should be tight. For example, a company producing reports in accordance with the Sarbanes-Oxley Act typically wishes to release no more information than is required by law; thus, the information that their financial system is permitted to release should be identical to the information it is required to release. In other systems, the bounds are not tight, such as in a poker game where some players are permitted, but not required, to reveal their cards at the end of a hand.

The remainder of the paper is structured as follows. Section 2 uses the example of a simple credit card sales system to examine what it means for a system to satisfy required information release. Section 3 presents an interactive imperative language that we use in Section 4 to formally define required release. We also define *bounded release*, a security condition that specifies what information a program is required and permitted to release. We show in Section 5 that required release and bounded release can be soundly enforced in an interactive language by a type system. Section 6 discusses related work, and Section 7 concludes. Appendices A and B contain proofs of the key theorems.

2 What is required release?

Consider, as a running example, a (grossly simplified) credit card sales system that takes a credit card number as input from high confidentiality channel H, and is required to release the last four digits to low confidentiality channel L (representing, for example, the printer, or an audit log). What does it mean for this system to satisfy the required release of the last four digits?

Noninterference is a strong information security condition that requires that public outputs reveal no information about confidential inputs. Any system that releases confidential information violates noninterference; the credit card sales system, which must release the last four digits of the confidential credit card number to a publicly observable printer, violates noninterference. However, just because a system violates noninterference does not mean it satisfies the required information release.

Consider the following attempt to implement the credit card sales system.

P_1 : input cc from H;
if $(cc \bmod 10,000) < 5,000$ then
output 0 to L
else
output 1 to L

The program inputs the credit card number from channel H, and then outputs either 0 or 1 to channel L. The output observed on channel L is influenced by the last four digits of the confidential input, and thus the program does not satisfy noninterference. However, the program fails to satisfy the required information release, as an observer of channel L does not learn the credit card number's last four digits.

Even if a system outputs the information required for release, it may fail to satisfy the required information release. This is demonstrated in the following program, which is another attempt to implement the credit card sales system.

P_2 : input cc from H;
$i := 0$;
while $i < (cc \bmod 10,000)$ do
output i to L;
$i := i + 1$;
output $(cc \bmod 10,000)$ to L;
$i := i + 1$;
while $i < 10,000$ do
output i to L;
$i := i + 1$

The command output $(cc \bmod 10,000)$ to L in program P_2 above explicitly outputs the credit card's last four digits. However, every execution of the program outputs all integers from 0 to 9,999 in order. An observer of channel L always sees the same output, regardless of the credit card's last four digits, and so the observer learns nothing. (Indeed, program P_2 satisfies noninterference: the observer cannot learn anything about the credit card number.)

These examples show that it is insufficient for observable output to be merely correlated with the information required for release, or even for the output to contain that information. The key insight is that to satisfy required release, the output must allow an observer to know what information was required for release.

In models of knowledge based on possible world semantics [22, 17], an agent has *implicit knowledge* (or, simply, *knowledge*) of fact ϕ if in all possible worlds consistent with the agent's observations, ϕ is satisfied. In the credit card system, an observer of channel L knows the last four digits of the credit card if all credit cards that could have produced the observed output end in the same four digits. Programs P_1 and P_2 do not allow an observer of channel L to know the last four digits.

Standard logical approaches to knowledge suffer from the problem of logical omniscience: an agent knows all logical consequences of its knowledge. The following attempt to implement the credit card system demonstrates this problem. The

program chooses two large primes, outputs their product, and the result of XOR-ing the smaller prime with the last four digits of the credit card number (padded with random bits to be the same length as the prime).

$$
\begin{aligned}
P_3: \quad & \mathsf{input}\ cc\ \mathsf{from}\ H; \\
& p := generateLargePrime(); \\
& q := generateLargePrime(); \\
& \mathsf{output}\ p \times q\ \mathsf{to}\ L; \\
& \mathsf{if}\ p < q\ \mathsf{then}\ t := p\ \mathsf{else}\ t := q; \\
& \mathsf{output}\ t\ \mathsf{xor}\ pad(cc\ \mathsf{mod}\ 10,000)\ \mathsf{to}\ L
\end{aligned}
$$

A logically omniscient observer of the program's output knows what the last four digits of the credit card number are. However, determining this requires factoring a large number, which is beyond the abilities of humans and current computer systems to perform in reasonable time.

Algorithmic knowledge [20] was introduced to address the problem of logical omniscience, and we can use algorithmic knowledge to reason whether a system satisfies required release.

An agent has *algorithmic knowledge*, or *explicit knowledge*, of fact ϕ if the agent has an algorithm that responds "Yes" when given input ϕ and the agent's observations. The agent's knowledge algorithm is *sound* if whenever it responds "Yes" then the agent has implicit knowledge of ϕ, and whenever it responds "No" then the agent does not have implicit knowledge of ϕ. The knowledge algorithm may also respond "?" when it cannot determine whether the agent does or does not have implicit knowledge. A knowledge algorithm is *complete* if it always answers "Yes" or "No" and never answers "?".

Rich classes of knowledge algorithms have been studied that can conservatively overestimate the computational ability of agents without giving the agents logical omniscience (e.g., [37, 36]). However, we are interested in simple algorithms. Such algorithms may be described in user manuals, specified by a government agency or auditor, or may be inferred from self-explanatory output. In all cases, the aim is to make it easy for an observer to learn the released information. In this setting, the observer is not the adversary, and it is acceptable (even desirable) to *underestimate* the observer's computational abilities, much as an instruction manual aims to be usable by as wide an audience as possible. For some programs (such as P_3), there may be sound knowledge algorithms that are beyond the ability of any observer to execute in reasonable time; such programs do not allow the observer to easily learn the released information, and are thus of no interest to us. For required release, we are concerned with the existence of sufficiently simple sound knowledge algorithms.

The following program does release the last four digits of the credit card number to channel L.

$$
\begin{aligned}
P_4: \quad & \mathsf{input}\ cc\ \mathsf{from}\ H; \\
& \mathsf{output}\ \text{"Last 4 credit card digits: "}\ \mathsf{to}\ L; \\
& \mathsf{output}\ (cc\ \mathsf{mod}\ 10,000)\ \mathsf{to}\ L
\end{aligned}
$$

Moreover, there is a simple sound knowledge algorithm to provide explicit knowledge for an observer of channel L: given fact $\phi \equiv (cc\ \mathsf{mod}\ 10,000) = n$, respond "Yes" if and only if the second output is n. Because there is a simple sound algorithm, an observer can gain explicit (and implicit) knowledge of the last four credit card digits, and so the program satisfies the required

information release.

To specify required release, we must specify not only *what* information is to be released, but also *how* that information is to be learned. We formalize this intuition by defining required information release in terms of a simple interactive programming language.

3 Language

In this section we present a simple interactive imperative programming language due to O'Neill et al. [33]. We use an interactive language as it is more general than the batch model traditionally used to reason about language-based information flow, and it can more accurately model real world programs that interact with their external environment, such as server processes, and programs with user interfaces.

We assume set $\mathcal{L}$ of security levels, ordered by a reflexive transitive relation $\sqsubseteq$ that indicates the relative restrictiveness of the levels. In this paper, our examples use the two element set $\mathcal{L} = \{L, H\}$ where $L \sqsubseteq H$. Security level L represents low confidentiality, and security level H represents high confidentiality. More expressive security levels are possible (e.g., [31, 10]). Metavariable ℓ ranges over security levels.

3.1 Users, channels, and strategies

Users interact with executing programs. We assume that security levels characterize users: the security level of a user indicates the most restrictive level of information the user is permitted to read. We assume that users with the same security level can freely collaborate, and so, without loss of generality, assume only a single user at each level.

Users communicate with executing programs via *channels*. We assume input on channels is blocking, and output is non-blocking. We assume that there is a single channel for each user, which, given the assumption of a single user for each security level, implies a single channel for each security level. We thus identify channels with security levels. An *event* is the transmission of an input or output on a channel. Event $in(\ell, v)$ denotes the input of value v on channel ℓ, and event $out(\ell, v)$ denotes the output of value v on channel ℓ. For simplicity we restrict values to integers.

We use $\mathbf{Ev}_{in}$ and $\mathbf{Ev}_{out}$ to denote, respectively, the set of all input and output events. We use $\mathbf{Ev}(\ell)$ to denote the set of all events that could occur on channel ℓ, and $\mathbf{Ev}$ to denote the set of all events.

$$\begin{aligned}
\mathbf{Ev}_{in} &\triangleq \textstyle\bigcup_{\ell\in\mathcal{L},v\in\mathbb{Z}}\{in(\ell,v)\} \\
\mathbf{Ev}_{out} &\triangleq \textstyle\bigcup_{\ell\in\mathcal{L},v\in\mathbb{Z}}\{out(\ell,v)\} \\
\mathbf{Ev}(\ell) &\triangleq \textstyle\bigcup_{v\in\mathbb{Z}}\{in(\ell,v), out(\ell,v)\} \\
\mathbf{Ev} &\triangleq \textstyle\bigcup_{\ell\in\mathcal{L}} \mathbf{Ev}(\ell)
\end{aligned}$$

Given $E \subseteq \mathbf{Ev}$, an *event trace on* E is a finite or infinite sequence $t = \langle \alpha_0, \alpha_1, \ldots \rangle$ such that $\alpha_i \in E$ for all i such that $0 \leq i < |t|$, where $|t|$ is the length of t. For infinite traces t, we define $|t| = \infty$. The ith element of event trace t is denoted

$t(i)$, for i such that $0 \leq i < |t|$. The empty trace is denoted $\langle\rangle$. We write $t\hat{}t'$ for the concatenation of finite trace t and trace t'. For traces t and t', we say that t *extends* t', written $t \succeq t'$, when t' is a prefix of t. Note that if t is an infinite trace, then t is the only trace that extends it. The set of all traces on E is denoted $\mathbf{Tr}(E)$.

The *restriction of trace t to E*, written $t \restriction E$, is the trace obtained by removing from t all events not contained in E. We write $t \restriction \ell$ as shorthand for $t \restriction \mathbf{Ev}(\ell)$.

User strategies express the behavior of users by describing how users interact with a program. Given trace t, a user of a channel with security level ℓ observes the event trace $t \restriction \mathbf{Ev}(\ell)$; a user's observations may influence their subsequent interaction with the program. Formally, a user strategy for a channel with security level ℓ is a function of type $\mathbf{Tr}(\mathbf{Ev}(\ell)) \to \mathbb{Z}$, and expresses what input a user will provide given their previous observations.

Let **UserStrat** be the set of all user strategies. A *joint strategy* is a collection of user strategies, one for each channel. Formally, a joint strategy ω is a function of type $\mathcal{L} \to \mathbf{UserStrat}$, that is, a function from security levels to user strategies.

User strategies are sensitive information. In general, we want to ensure that lower security users do not learn about strategies employed by higher security users: user ℓ should not learn anything about the strategy of user ℓ', where $\ell' \not\sqsubseteq \ell$. However, information release will violate this, revealing some information about the strategies of higher security users. In Section 4 we will discuss security requirements, and formally define semantic security conditions.

3.2 Syntax and semantics

We use a simple imperative language, extended with input, output, and declassification commands. The syntax of this language is:

$$
\begin{array}{rl}
\text{(expressions)} & e ::= n \mid x \mid e_0 \oplus e_1 \\
\text{(commands)} & c ::= \mathsf{skip} \mid x := e \mid c_0; c_1 \mid \\
& \qquad \mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \mid \\
& \qquad \mathsf{while}\ e\ \mathsf{do}\ c \mid \\
& \qquad \mathsf{input}\ x\ \mathsf{from}\ \ell \mid \\
& \qquad \mathsf{output}\ e\ \mathsf{to}\ \ell \mid \\
& \qquad x := \mathsf{declassify}(e\ \mathsf{to}\ \ell)
\end{array}
$$

Metavariable x ranges over **Var**, the set of all program variables. Variables take integer values, and literal values n also range over integers. Metavariable $\oplus$ ranges over total binary operations on the integers. A *state* σ maps variables to values, and so is a function of type $\mathbf{Var} \to \mathbb{Z}$. A *configuration* is a 4-tuple (c, σ, t, ω) representing a system about to execute c with state σ and joint strategy ω. Finite trace t is the history of events produced by the system so far. Terminal configurations have the form $(\mathsf{skip}, \sigma, t, \omega)$. Metavariable m ranges over configurations.

We define a small-step operational semantics for our language, using the relation $\longrightarrow$ over configurations. If $(c, \sigma, t, \omega) \longrightarrow (c', \sigma', t', \omega)$ then execution of command c can take a single step to command c', while updating the state from σ to σ'. Trace t' extends t with any events that were produced during the step. Joint strategy ω is unchanged when a configuration takes a step,

and is included in configurations to simplify notation and presentation.

Fig. 1 presents inference rules for the operational semantics. We use $\sigma(e)$ to denote the evaluation of expression e where each variable x is replaced with the integer $\sigma(x)$. Input command $\mathsf{input}\ x\ \mathsf{from}\ \ell$ takes the next input value v as defined by the user strategy for ℓ, assigns it to variable x, and updates the trace with input event $in(\ell, v)$. Similarly, output command $\mathsf{output}\ e\ \mathsf{to}\ \ell$ evaluates e to v, and updates the trace with output event $out(\ell, v)$. Declassification $x := \mathsf{declassify}(e\ \mathsf{to}\ \ell)$ is semantically equivalent to assignment $x := e$; the declassify annotation and security level ℓ are used in the type system, described in Section 5.

We use $\longrightarrow^*$ to denote the reflexive transitive closure of $\longrightarrow$. For finite trace t, we say configuration m *emits* t, written $m \rightsquigarrow t$, if there is some configuration (c, σ, t, ω) such that $m \longrightarrow^* (c, \sigma, t, \omega)$. For infinite trace t, m emits t if m emits all finite prefixes of t. Note that emitted events may include both input and output events.

4 Security definitions

In this section we define the security conditions *weak required release* and *strong required release*, which formally express what it means for a program to satisfy the required release of information. We also present the security conditions *noninterference* (e.g., [19, 38, 2]) and *delimited release* [39]. Noninterference requires that a program does not release any confidential information. Delimited release weakens noninterference by specifying what confidential information a program is allowed to release. We combine delimited release and required release to define bounds on what a program is permitted and required to release.

4.1 Required release

To formally define required release, we must be able to express *what* information is to be released, and *how* that information is to be learned by an observer. We introduce *input expressions* and *output expressions* to express each of these respectively. Input expressions are expressions over input values supplied on channels; output expressions are expressions over values output on a single channel.

The syntax for input and output expressions is:

$$\text{(input expressions)} \quad f ::= n \mid f_0 \oplus f_1 \mid \mathsf{in}_\ell[i]$$
$$\text{(output expressions)} \quad g ::= n \mid g_0 \oplus g_1 \mid \mathsf{out}[i]$$

Input expression $\mathsf{in}_\ell[i]$ refers to the ith input event on channel ℓ, for $i \in \mathbb{N}$. Input expressions may also contain integer constants and binary operations. Input expressions are evaluated against a trace. The judgment $t \vDash_{in} f \Downarrow v$ means that with trace t, input expression f evaluates to value v. Evaluation rules for input expressions are given in Fig. 2. If t does not have an ith input event on channel ℓ, then $\mathsf{in}_\ell[i]$ evaluates to $\bot$, that is, $t \vDash_{in} \mathsf{in}_\ell[i] \Downarrow \bot$. We assume that any binary operator $\oplus$ defined is total over $\mathbb{Z}_\bot$ and strict, meaning that for all $m, n \in \mathbb{Z}_\bot$, $m \oplus n$ is defined, and if m or n is $\bot$, then $m \oplus n = \bot$.

Output expressions are also evaluated against a trace. The judgment $t \vDash^{\ell}_{out} g \Downarrow v$ means that output expression g evaluates to value v using trace t restricted to channel ℓ events. Output expression $\mathsf{out}[i]$ refers to the ith output event on channel ℓ, for $i \in \mathbb{N}$. Fig. 2 also presents the evaluation rules for output expressions. Similar to input expressions, if there is no ith output event on channel ℓ, then $\mathsf{out}[i]$ evaluates to $\bot$.

Note that output expressions $\mathsf{out}[i]$ do not have a subscript indicating which channel ℓ the output occurs on. This is because output expressions are intended to be evaluated by a single user ℓ, using only the output events on that user's channel.

Intuitively, user ℓ learns input expression f from command c using output expression g, if in every execution that g evaluates to an integer value (using the output provided to ℓ), then f evaluates to the same integer. Thus, input expression f indicates *what* information the user is to learn, and output expression g indicates *how* the user learns it—g provides a sound knowledge algorithm. This leads us to our first definition of required release.

Command c satisfies *weak required release* of input expression f to user ℓ using output expression g if for any trace t that can be emitted by executing c, if t provides enough output to ℓ to evaluate g, then f and g evaluate to the same value.

Definition 1 (Weak required release)**.** Command c satisfies *weak required release* of input expression f to user ℓ using output expression g exactly when:

$$\begin{gathered}\text{For all configurations } m = (c, \sigma, \langle\rangle, \omega)\\ \text{and for all traces } t \text{ such that } m \rightsquigarrow t,\\ \text{if } t \vDash^{\ell}_{out} g \Downarrow v \text{ and } v \neq \bot \text{ then } t \vDash_{in} f \Downarrow v.\end{gathered}$$

Program P_4 satisfies weak required release of $\mathsf{in}_H[0] \bmod 10,000$ to L using $\mathsf{out}[1]$: the second output to L is the last four digits of the first H input (the credit card number). By contrast, programs P_1 and P_2 do not satisfy weak required release of $\mathsf{in}_H[0] \bmod 10,000$ to L for any output expression.

Weak required release is "weak" in that there is no requirement that command c provide sufficient output to ℓ for g to evaluate to an integer value. For example, the program skip satisfies weak required release of any input expression to L using output expression $\mathsf{out}[0]$, since no output is ever given to L, and output expression $\mathsf{out}[0]$ never evaluates to an integer value.

We can strengthen weak required release to ensure that command c always eventually provides sufficient output to ℓ for g to evaluate to an integer value. Command c satisfies *strong required release* of input expression f to user ℓ using output expression g if for any trace t that can be emitted by executing c, there is a trace t' that extends t, can be emitted by executing c, and provides sufficient output to ℓ to evaluate g, and f and g evaluate to the same value.[1]

Definition 2 (Strong required release)**.** Command c satisfies *strong required release* of input expression f to user ℓ using output expression g exactly when:

$$\begin{gathered}\text{For all configurations } m = (c, \sigma, \langle\rangle, \omega)\\ \text{and for all traces } t \text{ such that } m \rightsquigarrow t,\\ \text{there exists trace } t' \text{ such that } t' \succeq t,\ m \rightsquigarrow t', \text{ and}\\ t' \vDash^{\ell}_{out} g \Downarrow v \text{ and } t' \vDash_{in} f \Downarrow v \text{ for some } v \neq \bot.\end{gathered}$$

[1] Since the language is deterministic, this definition suffices to ensure that enough output is always eventually produced; the definition would need to be modified for non-deterministic and probabilistic languages.

Provided that input expression f depends nontrivially on every subexpression of the form $\mathsf{in}_\ell[i]$, strong required release is implies weak required release: if command c satisfies strong required release of f to ℓ using g, then it satisfies weak required release of f to ℓ using g.

More formally, we say that input expression f depends nontrivially on subexpression $\mathsf{in}_\ell[i]$ if given any two traces t and t' that are identical except for the value of the ith input event on channel ℓ, and $t \vDash_{in} f \Downarrow v$ and $t' \vDash_{in} f \Downarrow v'$ for some $v \neq \bot$ and $v' \neq \bot$, then $v \neq v'$.

For the remainder of the article, we restrict our attention to input expressions that depend nontrivially on every subexpression of the form $\mathsf{in}_\ell[i]$.

Theorem 1. *If command c satisfies strong required release of input expression f to user ℓ using output expression g, and f depends nontrivially on every subexpression of the form $\mathsf{in}_{\ell'}[i]$, then it satisfies weak required release of input expression f to user ℓ using output expression g.*

A proof of Theorem 2 appears in Appendix A.

The following program satisfies weak required release, but not strong required release of $\mathsf{in}_H[0] \bmod 10,000$ to L using $\mathsf{out}[1]$, because in some cases it will never produce sufficient output to L. (For presentation purposes, we assume that constant strings, such as "Last 4 credit card digits: " can be converted to appropriate constant integer values, and output to channels.)

$$
\begin{array}{ll}
P_5: & \mathsf{input}\ cc\ \mathsf{from}\ H; \\
& \mathsf{output}\ \text{“Last 4 credit card digits: ”}\ \mathsf{to}\ L; \\
& \mathsf{if}\ cc \bmod 10 = 0\ \mathsf{then}\ (\mathsf{while}\ 1\ \mathsf{do}\ \mathsf{skip})\ \mathsf{else}\ \mathsf{skip}; \\
& \mathsf{output}\ (cc \bmod 10,000)\ \mathsf{to}\ L
\end{array}
$$

Program P_4 satisfies strong required release of $\mathsf{in}_H[0] \bmod 10,000$ to L using $\mathsf{out}[1]$, because it always produces appropriate output to channel L.

Connection to explicit knowledge If a program satisfies (weak or strong) required release of input expression f to user ℓ using output expression g, then output expression g provides a sound knowledge algorithm for ℓ to learn the value of f. The knowledge algorithm takes as input a formula ϕ and the sequence of events that ℓ has observed. The knowledge algorithm is straightforward:

> If $\phi \equiv f = n$ and $t \vDash^{\ell}_{out} g \Downarrow n$ then respond "Yes".
>
> Otherwise, respond "?".

Note that the algorithm never responds "No", and if the algorithm responds "Yes", then, because the program satisfies required release of f to ℓ using g, $t \vDash^{\ell}_{out} g \Downarrow n$ implies $f = n$. Thus, the knowledge algorithm is sound.

Strong and weak required release are both parameterized by output expression g. As discussed in Section 2, the output expression g may be specified by the consumer of the output (such as an auditor or government agency), an instruction manual, or may be described by the program's output (as in Program P_4, where the text "Last 4 credit card digits" is output just before the last four credit card digits). In practice, there may be additional requirements on the form of the output function, such as a

limit on the number of steps required to evaluate it (analogous to requiring that the instructions for a task be no more than two pages).

Confidentiality, integrity, availability, and properties Required information release is primarily concerned with the confidentiality of information: it is phrased in terms of what an observer must be able to learn about confidential inputs to the system. However, required information release is also related to the integrity and availability of information. Weak information release is an integrity requirement: if the output expression evaluates to an integer value, it must equal the evaluation of the input expression. Strong information release contains an availability requirement: the output expression must eventually evaluate to an integer value. Information security requirements are not always easily separable into confidentiality, integrity, and availability requirements.

Weak and strong required release can be defined as *properties*: predicates over single execution traces. Weak required release is a safety property, and strong required release is neither safety nor liveness [1]. Recent work by Clarkson and Schneider [14] expresses some information-flow conditions as *hyperproperties*: predicates of sets of traces. They note that all information-flow conditions they considered were hyperproperties and not properties. Although weak and strong required release are properties, they clearly constitute part of a system's information flow requirements, so some information-flow conditions of interest are properties. Indeed, in Section 4.3 below, we discuss the relationship between required release and delimited release, an information flow security condition for permitted information release that is a hyperproperty and not a property.

4.2 Noninterference

Noninterference is a well-known semantic security condition that requires that public observations reveal no secrets. While there are many definitions of noninterference (e.g., [19, 38]), the most relevant for our purposes are termination- and progress-insensitive definitions for interactive models (e.g., [33, 5, 4]). Applied to the interactive setting used here, noninterference ensures that user ℓ does not gain any knowledge about the strategy employed by any user ℓ' such that $\ell' \not\sqsubseteq \ell$. That is, the strategy of any such user ℓ' does not interfere with the trace observed by user ℓ.

More precisely, a program satisfies noninterference if, for all security levels ℓ, and all configurations m and m' that agree on the user strategies of all users ℓ' such that $\ell' \sqsubseteq \ell$, the traces emitted by m and m' are indistinguishable to user ℓ. Two traces t and t' are *indistinguishable* to user ℓ, written $t \approx_\ell t'$ if $t \restriction \ell$ extends $t' \restriction \ell$, or vice-versa.

Definition 3 (Noninterference)**.** A command c satisfies *noninterference* exactly when for all levels $\ell \in \mathcal{L}$:

$$
\begin{aligned}
&\text{For all } m = (c, \sigma, \langle\rangle, \omega) \text{ and } m' = (c, \sigma, \langle\rangle, \omega') \\
&\quad \text{such that } \omega(\ell') = \omega'(\ell') \text{ for all } \ell' \sqsubseteq \ell, \\
&\text{and for all traces } t, t' \text{ such that } m \rightsquigarrow t \text{ and } m' \rightsquigarrow t', \\
&\text{we have } t \approx_\ell t'.
\end{aligned}
$$

This definition of noninterference generalizes that of O'Neill et al. [33] for arbitrary sets of security levels $\mathcal{L}$, and weakens it to be both termination insensitive and progress insensitive. The definition of trace indistinguishability used here is suitable

given the observational model, which does not allow a user to directly distinguish a terminated program from a program in a non-terminating loop (termination insensitivity), or from a program that may eventually produce additional output (progress insensitivity).

Note that strong required release of f to ℓ violates noninterference if the input expression f contains an input expression $\mathsf{in}_{\ell'}[i]$ such that $\ell' \not\sqsubseteq \ell$ (and the evaluation of f depends nontrivially on $\mathsf{in}_{\ell'}[i]$). For example, any program that satisfies strong required release of $\mathsf{in}_H[0] \bmod 10,000$ to L (such as program P_4) must violate noninterference, since $H \not\sqsubseteq L$, and L learns something about the strategy of H, to wit, the last four digits of the credit card number that H entered.

Weak required release of f to ℓ does not necessarily violate noninterference if the input expression f contains an input expression $\mathsf{in}_{\ell'}[i]$ such that $\ell' \not\sqsubseteq \ell$. However, if a program satisfies both noninterference and weak required release for such an input expression, then the program never produces sufficient output to evaluate the output expression.

4.3 Delimited and bounded release

Noninterference is a very restrictive security condition. Many real-world programs must violate noninterference in order to satisfy functional requirements that require or allow the release of information.

The security condition *delimited release* [39] weakens noninterference by specifying what information a program is permitted to release.

An *escape hatch* is a pair (f, ℓ) of input expression f, and security level ℓ. Intuitively, given escape hatch (f, ℓ), a program is permitted to release information f to security level ℓ.[2] Thus, given escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$, user ℓ is permitted to learn the evaluation of f_i for any escape hatch (f_i, ℓ_i) such that $\ell_i \sqsubseteq \ell$, in addition to the user strategies of any user ℓ' such that $\ell' \sqsubseteq \ell$.

A program satisfies delimited release by escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$ if, for any security level ℓ and configurations m and m' that have the same user strategy for any user ℓ' such that $\ell' \sqsubseteq \ell$, if m and m' respectively emit traces t and t' that agree on the evaluation of all escape hatches that may release information to level ℓ, then the traces emitted by m and m' are indistinguishable to user ℓ. Formally, we say that traces t and t' *agree up to ℓ on escape hatches* $(f_0, \ell_0), \ldots, (f_k, \ell_k)$ if for all $i \in 0..k$ such that $\ell_i \sqsubseteq \ell$, we have $t \vDash_{in} f_i \Downarrow v_i$ and $t' \vDash_{in} f_i \Downarrow v_i$ for some $v_i \neq \bot$.

Definition 4 (Delimited release)**.** Command c satisfies *delimited release* by escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$ exactly when for all levels $\ell \in \mathcal{L}$:

For all $m = (c, \sigma, \langle\rangle, \omega)$ and $m' = (c, \sigma, \langle\rangle, \omega')$
  such that $\omega(\ell') = \omega'(\ell')$ for all $\ell' \sqsubseteq \ell$,
and for all traces t, t' such that $m \rightsquigarrow t$ and $m' \rightsquigarrow t'$,
if t and t' agree up to ℓ on
    escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$,
then $t \approx_\ell t'$.

[2] Sabelfeld and Myers [39] specify escape hatches as declassification expressions $\mathsf{declassify}(e \text{ to } \ell)$, and expressions in escape hatches refer to initial values of variables.

Delimited release generalizes noninterference: if command c satisfies delimited release by an empty set of escape hatches, then c satisfies noninterference.

Both delimited release and required release are concerned with information flow, and with the knowledge an observer acquires. Required release specifies what information, at a minimum, a program must release. It specifies what an observer must be able to (explicitly) know, and can be viewed as specifying "lower bounds" on what information a program releases. By contrast, delimited release specifies what an observer is permitted to (implicitly) know, and can be seen as specifying "upper bounds", the maximum information a program is permitted to release. We can combine the security conditions of delimited release and required release to obtain both upper and lower bounds on a program's information release.

By analogy with escape hatches (which are openings in the roof of a building, and specify the upper bounds on information release), we use escape chutes (passages through which objects move by means of gravity) to define the lower bounds of information release. An *escape chute* is a tuple (f, ℓ, g) of input expression f, security level ℓ, and output expression g. We define bounded release by combining delimited release by escape hatches, and required release by escape chutes.

Definition 5 (Bounded release)**.** Command c satisfies *weak (strong) bounded release* by escape chutes

$$(f_0, \ell_0, g_0), \ldots, (f_k, \ell_k, g_k)$$

and escape hatches

$$(f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)$$

exactly when

1. for all $i \in 0..k$, c satisfies weak (strong) required release of f_i to ℓ_i using g_i; and
2. c satisfies delimited release by escape hatches $(f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)$

Program P_4 satisfies strong bounded release by escape chute $(\mathsf{in}_H[0]\mathsf{mod}10,000, L, \mathsf{out}[1])$ and escape hatch $(\mathsf{in}_H[0]\mathsf{mod}10,000, L)$. Thus, bounded release tells us not only that P_4 releases the input expression $\mathsf{in}_H[0] \bmod 10,000$, but also that this is the *only* information released by P_4.

The following program has different upper and lower bounds. It satisfies strong bounded release by escape chute $(\mathsf{in}_H[0]\mathsf{mod}10,000, L, \mathsf{out}[1])$ and escape hatches $(\mathsf{in}_H[0] \bmod 10,000, L)$ and $(\mathsf{in}_H[0] \operatorname{div} 10^{15}, L)$. It always releases the last four digits of the credit card number (via output expression $\mathsf{out}[1]$) and it may in addition release information about the first digit of the (16 digit) credit card number.

P_6 : input cc from H;
$x :=$ declassify(cc mod $10,000$ to L);
output "Last 4 credit card digits: " to L;
output x to L;
$y :=$ declassify(cc div 10^{15} to L);
if $y = 4$ then output "Visa" to L else skip

There is a consistency property between the escape hatches and escape chutes. Since escape chutes are the "lower bounds" of information release, they must contain no more information than the escape hatches, the "upper bounds" of information release. More precisely, if t and t' are traces that can be produced by a command satisfying bounded release, and t and t' agree on all input and output events on all channels up to some level ℓ, and agree on the value of all escape hatches that declassify to ℓ or below, then for each escape chute at level ℓ or below, either t and t' agree on the value of the escape chute, or t and t' do not have sufficient input events to evaluate the escape chute. We say that traces t and t' *agree on escape chute* (f_i, ℓ_i, g_i) if $t \vDash_{in} f_i \Downarrow v_i$ and $t' \vDash_{in} f_i \Downarrow v_i$ for some $v_i \neq \bot$.

Property 1 (Consistency)**.** If command c satisfies (weak or strong) bounded release by escape chutes $(f_0, \ell_0, g_0), \ldots, (f_k, \ell_k, g_k)$ and escape hatches $(f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)$ then

for all $\ell \in \mathcal{L}$, and for all configurations $m = (c, \sigma, \langle\rangle, \omega)$
and $m' = (c, \sigma, \langle\rangle, \omega')$, and for all traces t, t' such that
$m \rightsquigarrow t$ and $m' \rightsquigarrow t'$,
if t and t' agree up to ℓ on
escape hatches $(f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)$, and
for all $\ell' \sqsubseteq \ell$ we have $t \restriction \ell' = t' \restriction \ell'$
then for all $i \in 0..k$ such that $\ell_i \sqsubseteq \ell$,
either $t \vDash_{in} f_i \Downarrow \bot$, or $t' \vDash_{in} f_i \Downarrow \bot$, or
t and t' agree on escape chute (f_i, ℓ_i, g_i).

Proof. (Sketch) Given $m = (c, \sigma, \langle\rangle, \omega)$, $m' = (c, \sigma, \langle\rangle, \omega')$, ℓ and t and t' such that t and t' agree up to ℓ on escape hatches $(f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)$, and $t \restriction \ell' = t' \restriction \ell'$ for all $\ell' \sqsubseteq \ell$, then we can construct joint user strategies ω_0 and ω'_0 such that $m_0 = (c, \sigma, \langle\rangle, \omega_0)$, $m'_0 = (c, \sigma, \langle\rangle, \omega'_0)$, and $m_0 \rightsquigarrow t$ and $m'_0 \rightsquigarrow t'$, and $\omega_0(\ell') = \omega'_0(\ell')$ for all $\ell' \sqsubseteq \ell$.

For any escape chute (f_i, ℓ_i, g_i) such that $\ell_i \sqsubseteq \ell$, suppose $t \vDash^{\ell_i}_{out} g_i \Downarrow v_i$ and $t' \vDash^{\ell_i}_{out} g_i \Downarrow v'_i$ for some $v_i, v'_i \neq \bot$. By delimited release, $t \approx_{\ell_i} t'$, and so, t and t' agree on the values of all output expressions required to evaluate g_i to an integer value. Therefore, $v_i = v'_i$. By bounded release, the evaluation of f_i in t and t' also equal v_i, and so and t and t' agree on escape chute (f_i, ℓ_i, g_i). □

5 Enforcement

In this section we show that weak bounded release can be soundly enforced with a security type system. Weak bounded release is the conjunction of weak required release, and delimited release. Since weak required release is a safety property, clearly other enforcement mechanisms could also be used to enforce it, including dynamic mechanisms such as execution monitors. However, due to the similarity of escape chutes and escape hatches, a type system that enforces delimited release can easily be adapted to enforce weak bounded release as well.

Our type system conservatively tracks both the security level of information as it flows through a program, and what input expressions have been output and declassified. This allows us to ensure that (i) confidential information is never output to non-confidential channels; (ii) only appropriate escape hatches are declassified; and (iii) appropriate escape chutes are output to the correct channel in the correct order.

For command c, type judgments have the form

$$pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$$

where entities to the left of the turnstile ($\vdash$) indicate the context before the execution of c, and primed versions on the right hand side of the judgment indicate how the contexts change as a result of the execution of c.

Security level typing context Γ maps variables to security levels, and indicates an upper bound on the information stored in each variable. Program counter level pc is an upper bound on the information that may cause command c to be executed, and is used to track *implicit information flows* [15]. Typing context Γ and program counter level pc are standard in security type systems. Our type system is flow-sensitive, as it allows command c to modify Γ, and is based on the flow-sensitive security type system of Hunt and Sands [23].

The remaining entities in the context (C, Δ, E, and H) are used to track what input expressions have been output and declassified. Specifically, we conservatively track how many input and output events have been produced on each channel, what input expression (if any) is stored in each variable, what input expressions (if any) have been output, and what input expressions have been declassified.

- $C : \mathcal{L} \to \mathbb{Z}_\bot \times \mathbb{Z}_\bot$ counts the input and output events that have occurred on each channel. If $C(\ell) = (i, j)$, then the program has received i input events from channel ℓ, and produced j output events to channel ℓ. If $i = \bot$, then an unknown number of input events have been received on channel ℓ, and similarly, if $j = \bot$, an unknown number of output events have been produced.

- $\Delta : \textbf{Var} \to \textbf{InputExp}_\bot$ indicates what input expression is stored in each variable. For any variable x, if $\Delta(x) = f$ then the value stored in x is equivalent to input expression f. If $\Delta(x) = \bot$ then nothing is known about the value stored in x.

- $\mathrm{E} : \mathcal{L} \times \mathbb{Z} \to \textbf{InputExp}_\bot$ indicates which input expressions have been output to channels. If $\mathrm{E}(\ell, i) = f$ then the ith value output on channel ℓ was equal to the evaluation of input expression f. If $\mathrm{E}(\ell, i) = \bot$ then either the ith output to

channel ℓ has not yet been produced, or nothing is known about the ith output to channel ℓ.

- $H : \wp(\textbf{InputExp} \times \mathcal{L})$ is a set of escape hatches that may have been declassified.

Fig. 3 presents inference rules for the typing judgment. Given a function h, we write $h[a \mapsto b]$ for the function that evaluates to b on input a, and otherwise behaves like h. We use $\Gamma(e)$ to denote an upper bound of all levels $\Gamma(x)$ for variables x occurring in e; if $\mathcal{L}$ is a join semi-lattice, then this is the join of all $\Gamma(x)$ for x in e. We extend function Δ to a homomorphism on program expressions, and write $\Delta(e)$ for the result of applying the homomorphism to expression e.

In the typing rules, security level context Γ and program counter level pc do not interact with other parts of the context, and by themselves form a standard flow-sensitive information-flow security-type system, similar to that of Hunt and Sands [23]. In the following discussion of the typing rules, we focus on the type system's novel aspects.

For assignment $x := e$, T-ASSIGN updates input expression context Δ for x to $\Delta(e)$, which is either $\perp$ or an input expression equal to e at this program point. The typing rule T-DECLASSIFY for declassification $x := \mathsf{declassify}(e \text{ to } \ell)$ is similar to assignment, but escape hatch $(\Delta(e), \ell)$ is added to declassification effect H. Note that the rule implicitly requires $\Delta(e) \neq \perp$ since H must contain escape hatches. Rule T-SEQ simply threads the context through a sequence of commands. A skip command has no effect on the context, shown in rule T-SKIP.

Command $\mathsf{input}\ x\ \mathsf{from}\ \ell$ assigns the next input from channel ℓ to variable x. Rule T-IN updates input expression context Δ using helper function $\mathit{recordInput}(\Delta, x, C, \ell)$, which updates $\Delta(x)$ either to $\perp$ if the number of input events on channel ℓ is not known, or to input expression $\mathsf{in}_\ell[i]$, where i is the number of input events received on channel ℓ. If known, the number of input events on channel ℓ is incremented using the helper function $\mathit{inc}_{in}(C, \ell)$.

Command $\mathsf{output}\ e\ \mathsf{to}\ \ell$ outputs expression e to channel ℓ. Using helper function $\mathit{recordOutput}(\mathrm{E}, C, \ell, f)$, rule T-OUT records that the jth output on channel ℓ is equal to input expression $\Delta(e)$, where j is the number of output events produced on channel ℓ, and increments the number of output events produced on channel ℓ with helper function $\mathit{inc}_{out}(C, \ell)$. If the number of output events produced on channel ℓ is unknown (i.e., $j = \perp$), then no update to E or C is made.

The subsumption rule T-SUB allows the context to be weakened, or made less precise. It uses the flat ordering $\succeq$: for any lifted set $\mathbf{S}_\perp$, and for any $a, b \in \mathbf{S}_\perp$, $a \succeq b$ iff $a = b$ or $b = \perp$. We extend the $\succeq$ relation in the obvious way to pairs, and to a pointwise relation over functions. For example, $\Delta_0 \succeq \Delta_1$ iff for all $x \in \textbf{Var}$, $\Delta_0(x) \succeq \Delta_1(x)$. Similarly, we extend the binary relation $\sqsubseteq$ over $\mathcal{L}$ to a pointwise relation over functions with codomain $\mathcal{L}$.

The rules for if and while commands (T-IF and T-WHILE respectively) type check their sub-commands with a program counter level bounded below by pc and $\Gamma(e)$, since e controls the execution of the sub-commands. Rule T-WHILE requires that context is unchanged by the execution of the while command; for any channel ℓ, this means either that the loop body performs no input or output on ℓ, or that the context cannot precisely track the number on inputs or outputs received on channel ℓ, i.e., $C(\ell) = (i, j)$ and $\perp \in \{i, j\}$. Similarly for an if command, the context will lose track of the number on inputs or outputs received on channel ℓ unless both branches always perform the same number of inputs and outputs on ℓ.

The type system can easily be converted into an algorithmic type system, using the same technique as Hunt and Sands [23]. If the security levels $\mathcal{L}$ and binary relation $\sqsubseteq$ form a join-semi lattice, then type checking and type inference with the

algorithmic type system is decidable in time polynomial in the size of the program.

If command c is well-typed, then it satisfies both weak required release, and delimited release. Theorem 2 below states this claim formally. To state Theorem 2 concisely, we first introduce a helper function and additional notation.

Helper function $substOutExp(\mathrm{E}, \ell, g)$ takes output context E, security level ℓ, and output expression g, and substitutes any occurrence of $\mathsf{out}[i]$ with input expression $\mathrm{E}(\ell, i)$, that is, the input expression that E claims was the ith output on channel ℓ. For example, if $\mathrm{E}((L, 2)) = \mathsf{in}_H[1]$, then $substOutExp(\mathrm{E}, L, 42 + \mathsf{out}[2]) = 42 + \mathsf{in}_H[1]$. Rules for $substOutExp(\mathrm{E}, \ell, g)$ are given in Fig. 4.

We assume there is a notion of equivalence between input expressions, denoted by $\equiv$. We require that if $f_0 \equiv f_1$, then for all traces t and $v \in \mathbb{Z}_\perp$, $t \vDash_{in} f_0 \Downarrow v$ iff $t \vDash_{in} f_1 \Downarrow v$. The equivalence relation could be syntactic identity, or syntactic identity up to commutativity and associativity of operators, or, (depending on the operators in the language) a deeper semantic equivalence.

Finally, for any set $\mathbf{S}$ and $v \in \mathbf{S}$, we use $\overline{v}$ as shorthand for a constant function that always returns v. For example, $\overline{(0,0)}$ is a function that always returns the pair $(0, 0)$.

Theorem 2. *If $pc, \Gamma_0; \overline{(0,0)}, \overline{\perp}, \overline{\perp}, \emptyset \vdash c \triangleright \Gamma; C, \Delta, \mathrm{E}, H$ for some Γ_0 and pc, then*

1. *c satisfies weak required release of input expression f to user ℓ using output expression g if $substOutExp(\mathrm{E}, \ell, g) \equiv f$.*
2. *c satisfies delimited release by escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$ if for all $(f, \ell) \in H$ there exists $i \in 0..k$ such that $f \equiv f_i$ and $\ell_i \sqsubseteq \ell$.*

A proof of Theorem 2 appears in Appendix B.

If command c is well-typed, then because it satisfies both weak required release, and delimited release, it satisfies weak bounded release.

Corollary 1. *Command c satisfies weak bounded release by escape chutes $(f_0, \ell_0, g_0), \ldots, (f_k, \ell_k, g_k)$ and escape hatches $(f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)$ if*

$$pc, \Gamma_0; \overline{(0,0)}, \overline{\perp}, \overline{\perp}, \emptyset \vdash c \triangleright \Gamma; C, \Delta, \mathrm{E}, H$$

for some Γ_0 and pc, and $substOutExp(\mathrm{E}, \ell, g_i) \equiv f_i$ for all $i \in 0..k$ and for all $(f, \ell) \in H$ there exists $i \in 0..n$ such that $f \equiv f'_i$ and $\ell'_i \sqsubseteq \ell$.

Proof. Immediate from Theorem 2. □

Although program P_4 satisfies bounded release, it does not type-check: it attempts to release information from H to L but does not have any declassify annotations. Program P_6 does type-check. The judgment

$$L, \overline{L}; \overline{(0,0)}, \overline{\perp}, \overline{\perp}, \emptyset \vdash P_6 \triangleright \Gamma; C, \Delta, \mathrm{E}, H$$

holds for

$$
\begin{aligned}
\Gamma &= \overline{L}[cc \mapsto H, x \mapsto L, y \mapsto L] \\
C &= \overline{(0,0)}[H \mapsto (1,0), L \mapsto (0,\bot)] \\
\Delta &= \overline{\bot}[cc \mapsto \mathsf{in}_H[0], x \mapsto \mathsf{in}_H[0] \bmod 10,000, \\
&\qquad\qquad y \mapsto \mathsf{in}_H[0] \operatorname{\mathsf{div}} 10^{15}] \\
\mathrm{E} &= \overline{\bot}[(L,0) \mapsto \text{“...”}, (L,1) \mapsto \mathsf{in}_H[0] \bmod 10,000] \\
H &= \{(\mathsf{in}_H[0] \bmod 10,000, L), (\mathsf{in}_H[0] \operatorname{\mathsf{div}} 10^{15}, L)\}.
\end{aligned}
$$

Thus, by Corollary 1, P_6 satisfies weak bounded release by escape chute $(\mathsf{in}_H[0] \bmod 10,000, L, \mathsf{out}[1])$ and escape hatches $(\mathsf{in}_H[0] \bmod 10,000, L)$ and $(\mathsf{in}_H[0] \operatorname{\mathsf{div}} 10^{15}, L)$.

We have used a single type system to enforce weak bounded release, which consists of both delimited release and weak bounded release. It is worth noting which elements of the type system are used to enforce each of the two semantic security conditions. Enforcement of weak delimited release relies on C (count of input and output events), Δ (which input expression is stored in each variable), and E (which input expressions have been output to channels). By contrast, enforcement of delimited release relies on Γ (security level of each variable), pc (the program counter level, used to track implicit information flows), H (which escape hatches have been declassified), C, and Δ.

There is considerable overlap in the enforcement mechanisms for the two semantic security conditions. The only element needed to enforce weak required release that is not needed to enforce delimited release is E, which records which input expressions have been output to channels.

A more sophisticated static analysis (or a more restrictive language) could enforce strong required release, by reasoning about the termination of loops, and the eventual production of outputs.

6 Related work

Permitted information release Much recent work has considered information release, but focuses on the specification and enforcement of *permitted* information release. To the best of our knowledge, this work is the first to address *required* information release.

Sabelfeld and Sands [40] present a survey of work on (permitted) information release, and introduce four *dimensions of declassification* that provide a categorization of semantic security conditions for information release: *what* information is released, *who* releases the information, *when* does the release happen, and *where* in the program (and *where* in the ordering of security levels) does the release occur.

These dimensions are also relevant to required information release. This work is primarily concerned with *what* information is required for release, expressed using input expressions. Strong required release relates to the *when* aspect: it mandates that information is eventually released, whereas weak required release places no requirements on when (if ever) information is released. Further connections between various dimensions of information release and required information release are waiting

to be explored. For example, required information release could be strengthened by placing stronger requirements on *when* a system must release information. Previous work on specifying the conditions under which information is permitted to be released (e.g., [12, 6]) may also be applicable to specifying when information is required to be released.

The most closely related work on permitted information release is the work on *delimited release* [39], which was presented in Section 4 and forms a key component of bounded release security condition. Delimited release was extended to *localized delimited release* [3] by restricting not only *what* information may be released, but *where* it may be released (at an appropriate declassify command). The type system used by Sabelfeld and Myers [39] to enforce delimited release also enforces localized delimited release, so we speculate that the type system used in this paper to enforce bounded release would also enforce an appropriately defined *localized bounded release*, where the delimited release component of bounded release is restricted to localized delimited release.

Lux and Mantel [29] generalize delimited release through the addition of *explicit reference points* to escape hatches. Escape hatches, as originally presented by Sabelfeld and Myers [39], allow the declassification of escape hatch's valuation only at the initial memory. Lux and Mantel allow declassification of valuations at arbitrary but explicit program points. This increases the expressiveness of escape hatches. In an interactive setting, it would permit, for example, the controlled release of an input that occurs in the body of a loop. Similarly, the expressiveness of escape chutes could be increased, permitting the expression of required release of inputs that occur in loop bodies.

Broberg and Sands present *flow locks* [7, 8, 9], a foundational mechanism for describing and controlling information flow. A flow lock controls information flow between security levels, potentially at fine granularity. A program may explicitly open or close a flow lock, enabling or disabling information flow. Flow locks are intended to form a core calculus of information flow, and many information flow semantic security conditions and enforcement mechanisms can be encoded using flow locks. However, flow locks are concerned with permitted information flow, and it is unclear what modifications to flow locks would be required to reason about required information flow.

Swamy and Hicks [42] specify information release policies as security automata [41], and enforce that programs adhere to the policies using a sophisticated type system. The release policies focus on specifying appropriate obligations that must be satisfied before information is released. We believe that with relatively minor extensions, the information release policies could be extended to specify when information must be released. Indeed, weak required release is a safety condition, and is enforceable by *edit automata* [27], which are a more general class of security automata than the information release polices of Swamy and Hicks.

Sabelfeld and Sands [40] also introduce several *prudent principles of declassification*. *Semantic consistency* requires that the security of a program depends only on extensional properties of the program, not intensional properties. This principle is directly applicable to required release, and is satisfied by weak and strong required release, and bounded release: semantically equivalent programs satisfy the same security conditions. The other principles (*conservativity*, *monotonicity of release*, *non-occlusion*, and *trailing attacks*) are not directly applicable to required release. Lux and Mantel [28] present additional prudent principles of declassification, which are also not directly applicable to required release.

Knowledge-based semantic security conditions The definition of required information release was inspired by algorithmic knowledge [20]. Other semantic security conditions have also recently been defined in terms of attacker knowledge. Askarov and Sabelfeld [2] use knowledge to define *gradual release*: an attacker's knowledge of secrets may become more precise only at specified declassification events. Gradual release restricts permitted information release, and as such it suffices to use implicit knowledge; since we are concerned with required release, we use algorithmic knowledge to ensure that knowledge can be obtained with reasonable resources. The use of algorithmic knowledge leads us to specify *how* an observer learns released information, in addition to *what* information they learn.

O'Neill [32] presents many information flow conditions in an epistemic framework, but doesn't consider algorithmic knowledge or required information release.

Enforcement techniques Since weak required release is a safety property, it could be enforced using dynamic techniques, such as execution monitors [41]. Section 5 demonstrated that a security-type system for delimited release could be easily extended to enforce weak bounded release, and we anticipate that information-flow monitors (e.g., [25, 4]) could similarly be extended to enforce weak required release.

Askarov and Sabelfeld [4] present semantic security conditions that generalize localized delimited release and gradual release, and enforce these conditions a combination of static and dynamic techniques in an interactive language. Their semantic security condition in essence indexes inputs and outputs from the most recent event. By contrast, the version of delimited release (and required release) presented here indexes input and output absolutely, from the beginning of program execution. The languages for input and output expressions could be adapted to use most-recent indexing. This would enable a more permissive type system: currently, when an input or output occurs in the body of a loop, the type system is unable to count accurately, and precision is lost. Most-recent indexing would allow the type system to regain precision after a loop body in which input or output occurs.

Obligations An *obligation* is a requirement on a system's or subject's future behavior. Obligations often arise when a subject is granted access to data, for example, if a subject is allowed to check data out of a repository, she is obliged to eventually check the data back in, possibly within some fixed time constraint. Previous work has studied both the specification (e.g., [34]) and analysis (e.g., [11, 21, 18, 16]) of obligations. Although obligations have been considered with respect to privacy (e.g., [30]), we believe this is the first work to consider obligatory information flow.

Required release can be viewed as an obligation on the system to release information in an appropriate form. Weak required release is a form of integrity: if appropriate output is produced, it must correctly represent the information to be released. Strong required release is an obligation that must eventually be satisfied: the system must always be able to eventually produce sufficient output. In many practical settings, it may be desirable to strengthen required release, and bound the time until sufficient output is produced. There are many possible ways of expressing time bounds, including elapsed wall-clock or system time from the beginning of execution, or from some event (e.g., arrival of sufficient input), or requiring the obligation is satisfied before some other event occurs. Instead of considering these stronger notions of required release, we have focused here on the

interaction between required and permitted information release. We expect that strengthening required release with time bounds will not change the relationship between required and permitted release, as, for example, expressed in Property 1.

Irwin et al. [24] define *accountability* for violation of an obligation. As currently expressed, it is the system that is responsible for required release of information (and, as such, Irwin et al. do not regard it as an obligation, which are, by their definition, unenforceable by the system). If the *who* dimension of information release [40] were incorporated into required release, it may become the responsibility of some security principal to ensure the release of information, fitting into the model of accountability.

7 Conclusion

As part of their correct functionality, many systems are required (not just permitted) to release information. This paper introduces the problem of required information release: specifying, reasoning about, and enforcing, the information security of systems that must release information.

We have defined semantic conditions for required information release. Inspired by work on algorithmic knowledge, the semantic conditions must specify both *what* information is to be released, and *how* that information is to be learned by an observer. Input expressions specify what information is to be released, and output expressions specify how an observer learns the information. A program satisfies weak required release of input expression f to user ℓ using output expression g if whenever user ℓ is able to evaluate g, then f evaluates to the same value. A program satisfies strong required release if it satisfies weak required release, and eventually produces sufficient output for user ℓ to evaluate g.

We investigated the relationship between a system's required and permitted information release, and defined *bounded release*, which combines required release with delimited release. Bounded release specifies upper and lower bounds on the information a system releases. For many systems, these bounds should be tight: the system should release all and only information it is required to release. We have shown that (weak) bounded release can be conservatively enforced by a type system.

Both weak and strong required release are properties: predicates over single execution traces. Noninterference, delimited release, and many other information security requirements, are hyperproperties, but not properties. One may thus be concerned whether required information release is an information security requirement. We believe that required information release, while a property, is clearly concerned with the flow of information in a system: it requires that, at a minimum, certain information flows to an observer. We have shown a connection between required information release and delimited release: whereas required information release specifies the minimum information flow from high security inputs to low security outputs that a system must satisfy, delimited release specifies maximum information flow. Thus, we believe that required information release is part of a system's information security requirements.

There is still much left to understand with respect to required information release. There are systems with information release requirements that cannot be expressed using the policies presented in this paper. For example, financial reports of a company should be released to all shareholders, not a subset; if Alice and Bob are the shareholders, the system must release

reports to Alice if and only if it releases that information to Bob. In terms of enforcing required information release, it may be impractical to explicitly specify the knowledge algorithm by which an observer may learn the released information; static analyses may allow the automatic discovery of the knowledge algorithm, thus reducing the burden of proving a system satisfies required information release.

To build trustworthy computer systems, it is important to understand and provably enforce a system's information security requirements. By introducing the concept of required information release, and providing mechanisms to specify and enforce these requirements, this work brings us closer to the goal of strong, end-to-end, application-specific information security.

Acknowledgements

We thank Andrew Myers for very useful discussions and feedback about this work. We also thank Michael Clarkson, Allan Friedman, Tyler Moore, Kevin O'Neill, Fred Schneider, and Jeff Vaughan for interesting and helpful discussions related to this work, and the anonymous reviewers for their useful comments. This research is supported by the National Science Foundation under Grant No. 1054172.

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[2] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 207–221. IEEE Computer Society, 2007.

[3] Aslan Askarov and Andrei Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 53–60. ACM Press, 2007.

[4] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, 2009.

[5] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, October 2008.

[6] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2008.

[7] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proceedings of the 15th European Symposium on Programming*, pages 180–196. Springer, 2006.

[8] Niklas Broberg and David Sands. Flow-sensitive semantics for dynamic information flow policies. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, June 2009.

[9] Niklas Broberg and David Sands. Paralocks – role-based information flow control and beyond. In *Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

[10] Hubie Chen and Stephen Chong. Owned policies for information security. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2004.

[11] Laurence Cholvy and Frédéric Cuppens. Analyzing consistency of security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 103–112. IEEE Computer Society, 1997.

[12] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM Press, October 2004.

[13] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science*, 112:149–166, January 2005.

[14] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, July 2008.

[15] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[16] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Obligations and their interaction with programs. In *Proceedings of the 12th European Symposium On Research In Computer Security*, volume 4734, pages 375–389, Berlin, September 2007. Springer.

[17] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, MA, 1995.

[18] Janice I. Glasgow and Glenn H. MacEwen. Obligation as the basis of integrity specification. In *Proceedings of the 2nd IEEE Computer Security Foundations Workshop*, pages 64–70. IEEE Computer Society, 1989.

[19] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, April 1982.

[20] Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Algorithmic knowledge. In *Proceedings of the 5th Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 255–266, March 1994.

[21] Manuel Hilty, David Basin, and Alexander Pretschner. On obligations. In *Proceedings of the 10th European Symposium On Research In Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 98–117, Berlin, 2005. Springer.

[22] Jaakko Hintikka. *Knowledge and Belief.* Cornell University Press, 1962.

[23] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 79–90. ACM Press, January 2006.

[24] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligations. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 134–143, New York, NY, USA, 2006. ACM.

[25] Gurvan Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 218–232, 2007.

[26] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Conference Record of the Thirty-Second Annual ACM Symposium on Principles of Programming Languages*. ACM Press, January 2005.

[27] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium On Research In Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373, September 2005.

[28] Alexander Lux and Heiko Mantel. Who can declassify? In *Proceedings of the Workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *LNCS*, pages 35–49. Springer, 2009.

[29] Alexander Lux and Heiko Mantel. Declassification with explicit reference points. In *14th European Symposium on Research in Computer Security*, volume 5789 of *LNCS*, pages 69–85. Springer, 2009.

[30] Marco Casassa Mont. A system to handle privacy obligations in enterprises. Technical Report HPL-2005-180, Hewlett-Packard, October 2005.

[31] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 186–197. IEEE Computer Society, May 1998.

[32] Kevin R. O'Neill. *Security and Anonymity in Interactive Systems*. PhD thesis, Cornell University, August 2006.

[33] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2006.

[34] Jaehong Park and Ravi Sandhu. The UCON ABC usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[35] François Pottier and Vincent Simonet. Information flow inference for ML. In *Conference Record of the Twenty-Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 319–330, 2002.

[36] Riccardo Pucella. Deductive algorithmic knowledge. *Journal of Logic and Computation*, 16(2):287–309, 2006.

[37] R. Ramanujam. View-based explicit knowledge. *Annals of Pure and Applied Logic*, 96:343–368, 1999.

[38] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[39] Andrei Sabelfeld and Andrew C. Myers. A model for delimited release. In *Proceedings of the 2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191. Springer-Verlag, 2004.

[40] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[41] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[42] Nikhil Swamy and Michael Hicks. Verified enforcement of automaton-based information release policies. In *Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security*. ACM Press, June 2008.

[43] Ron van der Meyden. What, indeed, is intransitive noninterference? In *Proceedings of the 12th European Symposium On Research In Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 235–250. Springer, September 2007.

A Proof of Theorem 1

In this appendix we prove Theorem 1.

We first prove a lemma that states that if a program satisfies strong required release of input expression f using output expression g and the program has produced enough output for g to evaluate to a value, then the trace must also have read enough input for f to evaluate to a value.

Lemma 1. *Let command c satisfy strong required release of input expression f to user ℓ using output expression g. Let f depend nontrivially on every subexpression of the form $\mathsf{in}_\ell[i]$. Let $m = (c, \sigma, \langle\rangle, \omega)$ and for some trace t such that $m \rightsquigarrow t$, we have $t \vDash^{\ell}_{out} g \Downarrow v$ such that $v \neq \bot$. Then $t \vDash_{in} f \Downarrow v$.*

Proof. Suppose that $t \vDash^{\ell}_{out} g \Downarrow v$ such that $v \neq \bot$, but $t \vDash_{in} f \Downarrow v'$ for $v' \neq v$. If $v' \neq \bot$, then the command cannot satisfy strong required release. So it must be the case that $v' = \bot$. Since by assumptions all binary operators are total and strict, it must be because the trace has not received sufficient input to evaluate some input expression $\mathsf{in}_\ell[i]$. By strong required release, we can extend t to a trace t' such that $t' \vDash_{in} f \Downarrow v$.

But we can construct a new joint strategy ω_0 that is identical to ω except that the user strategy for channel ℓ returns a different value for the ith input than would be returned by ω. Consider $m_0 = (c, \sigma, \langle\rangle, \omega_0)$. Clearly $m_0 \rightsquigarrow t$, since ω_0 behaves identically to ω except for the ith input on channel ℓ which has not yet occurred. By strong required release, we can extend to a trace t'' such that $t'' \vDash_{in} f \Downarrow v$. But this violates the assumption that f depends nontrivially $\mathsf{in}_\ell[i]$. □

To prove Theorem 1, let c satisfy strong required release of input expression f to user ℓ using output expression g, and for configuration $m = (c, \sigma, \langle\rangle, \omega)$ assume that $m \rightsquigarrow t$ and $t \vDash^{\ell}_{out} g \Downarrow v$ such that $v \neq \bot$. We need to show $t \vDash_{in} f \Downarrow v$. This follows immediately from Lemma 1.

B Proof of Theorem 2

In this appendix we prove Theorem 2. Section B.1 shows that a well-typed program satisfies required release, and Section B.2 shows that a well-typed program satisfies delimited release. We first introduce and prove some useful lemmas and theorems about the type system.

We say that a configuration (c, σ, t, ω) satisfies context C, Δ, E if the context C, Δ, E accurately reflects the configuration. That is, the entity C records how many input and output events have been received and sent on each channel, and must agree with trace t. Similarly, Δ records input expressions that are equivalent to values stored in the state, and the state σ and trace t must agree on these values. Finally, E records input expressions that are equivalent to output values, and t must satisfy these relationships.

Definition 6. We say that configuration (c, σ, t, ω) satisfies context C, Δ, E, written $C, \Delta, \mathrm{E}, \vDash (c, \sigma, t, \omega)$ if all of the following conditions hold.

1. For all $\ell \in \mathcal{L}$, let $C(\ell) = (i, j)$.
   (a) Either $i = \bot$ or $|t \restriction (\mathbf{Ev}_{in} \cap \mathbf{Ev}(\ell))| = i$; and
   (b) Either $j = \bot$ or $|t \restriction (\mathbf{Ev}_{out} \cap \mathbf{Ev}(\ell))| = j$.
2. For all $x \in \mathbf{Var}$, either $\Delta(x) = \bot$ or $t \vDash_{in} \Delta(x) \Downarrow \sigma(x)$.
3. For all $\ell \in \mathcal{L}$ and $i \in \mathbb{N}$, either $\mathrm{E}(\ell, i) = \bot$ or $t \vDash_{in} \mathrm{E}(\ell, i) \Downarrow v$ where $(t \restriction (\mathbf{Ev}(\ell) \cap \mathbf{Ev}_{out}))(i) = out(\ell, v)$.

The operational semantics preserve typings.

Lemma 2 (Type preservation)**.** *If*

$$C, \Delta, \mathrm{E}, \vDash (c, \sigma, t, \omega)$$

and

$$pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$$

and

$$(c, \sigma, t, \omega) \longrightarrow (c', \sigma', t', \omega),$$

then there exists C'', Δ'', E'', H'', Γ'' and pc'' such that

$$pc'', \Gamma''; C'', \Delta'', \mathrm{E}'', H'' \vdash c' \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$$

and

$$C'', \Delta'', \mathrm{E}'', \vDash (c', \sigma', t', \omega).$$

Proof. By induction on the typing judgment $pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$. □

The type system ensures that the count of input and output events on each channel can only increase, or lose precision.

Lemma 3. *Given $pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$ and $\ell \in \mathcal{L}$, let $C(\ell) = (i, j)$ and $C'(\ell) = (i', j')$. Either $i \neq \bot$ and $i' \neq \bot$ and $i \leq i'$ or $i' = \bot$. Also, either $j \neq \bot$ and $j' \neq \bot$ and $j \leq j'$ or $j' = \bot$.*

Proof. Proof by induction on $pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$. □

Similarly, the type system ensures that for typing context E (which records input expressions that are equivalent to output values), the post-context may be less precise than the pre-context, but otherwise agrees with it. That is, once the type system has recorded that a given input expression is equivalent to a given output value, the type system can not change it to a different input expression.

Lemma 4. *Given $pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$ and $\ell \in \mathcal{L}$, let $C(\ell) = (i, j)$. If $j = \bot$ then for all $k \in \mathbb{N}$, we have $\mathrm{E}(\ell, k) \succeq \mathrm{E}'(\ell, k)$. If $j \in \mathbb{N}$ then for all $0 \leq k < j$, we have $\mathrm{E}(\ell, k) \succeq \mathrm{E}'(\ell, k)$.*

Proof. By induction on the typing judgment $pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$, using Lemma 3 for the case for sequence. □

The substitution function $substOutExp(\mathrm{E}, \ell, g)$ is correct, in that if output expression g evaluates to a value v, then replacing all $\mathsf{out}[i]$ subexpressions with appropriate input expressions stored in E will result in an input expression that also evaluates to v.

Lemma 5. *If $C, \Delta, \mathrm{E}, \vDash (c, \sigma, t, \omega)$ and $t \vDash^{\ell}_{out} g \Downarrow v$ and $\mathrm{E}(\ell, i) \neq \bot$ for all i such that $\mathsf{out}[i]$ appears in g (that may affect the evaluation of g), then $t \vDash_{in} substOutExp(\mathrm{E}, \ell, g) \Downarrow v$*

Proof. By induction on $substOutExp(\mathrm{E}, \ell, g)$. The only interesting case is $g = \mathsf{out}[i]$ (where $\mathsf{out}[i]$ may affect the evaluation of the whole output expression). In that case, $substOutExp(\mathrm{E}, \ell, g) = \mathrm{E}(\ell, i) \neq \bot$. We have $(t \restriction (\mathbf{Ev}_{out} \cap \mathbf{Ev}(\ell)))[i] = out(\ell, v)$, and so, since $C, \Delta, \mathrm{E}, \vDash (c, \sigma, t, \omega)$, we have $t \vDash_{in} \mathrm{E}(\ell, i) \Downarrow v$. □

B.1 Required release

Having shown several useful lemmas, we are now ready to prove that well-typed programs satisfy required release.

Lemma 6. *If*

$$pc, \Gamma_0; \overline{(0, 0)}, \overline{\bot}, \overline{\bot}, \emptyset \vdash c \triangleright \Gamma; C, \Delta, \mathrm{E}, H$$

for some Γ_0 and pc, and $substOutExp(\mathrm{E}, \ell, g) \equiv f$, then c satisfies weak required release of input expression f to user ℓ using output expression g.

Proof. Assume $pc, \Gamma_0; \overline{(0,0)}, \overline{\bot}, \overline{\bot}, \emptyset \vdash c \triangleright \Gamma; C, \Delta, \mathrm{E}, H$ for some Γ_0 and pc, and $substOutExp(\mathrm{E}, \ell, g) \equiv f$. Let $m = (c, \sigma, \langle\rangle, \omega)$ for some σ and ω. Suppose that $m \longrightarrow^* (c', \sigma', t, \omega)$ and $t \vDash^{\ell}_{out} g \Downarrow v$ for some $v \neq \bot$. We need to show that $t \vDash_{in} f \Downarrow v$.

By repeated application of Lemma 2, there exists C', Δ', E', H', Γ' and pc' such that $pc', \Gamma'; C', \Delta', \mathrm{E}', H' \vdash c' \triangleright \Gamma; C, \Delta, \mathrm{E}, H$ and $C', \Delta', \mathrm{E}', \vDash (c', \sigma', t, \omega)$. Since $t \vDash^{\ell}_{out} g \Downarrow v$ for $v \neq \bot$, there have been sufficient output events to channel ℓ to evaluate g to a non-$\bot$ value. Since $C', \Delta', \mathrm{E}', \vDash (c', \sigma', t, \omega)$, we have $C'(\ell) = (i, j)$ and either $j = \bot$ or $j = |t \restriction (\mathbf{Ev}_{out} \cap \mathbf{Ev}(\ell))|$. Either way, by Lemma 4, we have $substOutExp(\mathrm{E}', \ell, g) = substOutExp(\mathrm{E}, \ell, g)$. Since $substOutExp(\mathrm{E}', \ell, g) \equiv f$, then $\mathrm{E}'(\ell, i) \neq \bot$ for all i such that $out(\ell, i)$ appears in g (that may affect the evaluation of g). By Lemma 5, $t \vDash_{in} substOutExp(\mathrm{E}', \ell, g) \Downarrow v$. From the definition of $\equiv$, we have $t \vDash_{in} f \Downarrow v$ as required. □

B.2 Delimited release

We prove that the type system enforces delimited release user a proof technique based on the technique of Pottier and Simonet [35] for showing noninterference in the ML programming language. We define a new language, IMPI^2, that can represent two executions of a program. We show that type preservation in IMPI^2 implies that the program satisfies delimited release. (For convenience, we use IMPI to refer to the interactive imperative language presented in Section 3.)

B.2.1 Syntax and semantics

The language IMPI^2 extends the interactive language with pair constructs for commands $(\!| c_1 \mid c_2 |\!)$, integers $(\!| v_1 \mid v_2 |\!)$, and events $(\!| \alpha_1 \mid \alpha_2 |\!)$. The pair constructs represent different commands, integers, and events that may arise in two different executions of a program. A command pair cannot be nested inside another command pair, but can otherwise appear nested at arbitrary depth. Integer pairs are used to represent different input values that may be provided by different user strategies, and to track how states differ in different executions of a program: user strategies in IMPI^2 are functions from (IMPI) traces to integers and integer pairs, and stores in IMPI^2 are functions from variables to integers and integer pairs. We introduce the special event void, and allow elements of event pairs to range over events and void. The constant void is used to indicate that an event occurred in only one of the two executions. We also allow input and output values to range over integer pairs and integers.

$$\begin{array}{ll} \text{(expressions)} & e ::= \ldots \mid (\!| v_1 \mid v_2 |\!) \\ \text{(commands)} & c ::= \ldots \mid (\!| c_1 \mid c_2 |\!) \end{array}$$

For an extended command c, let the projections $\lfloor c \rfloor_1$ and $\lfloor c \rfloor_2$ represent the two commands that c encodes. The projection functions satisfy $\lfloor (\!| c_1 \mid c_2 |\!) \rfloor_i = c_i$, and are homomorphisms on other commands. Similarly for integer pairs, $\lfloor (\!| v_1 \mid v_2 |\!) \rfloor_i = v_i$. The projection functions are extended to states, so that

$$\lfloor \sigma \rfloor_i(x) = \begin{cases} v & \text{if } \sigma(x) = n \\ v_i & \text{if } \sigma(x) = (\!| v_1 \mid v_2 |\!) \end{cases}$$

The evaluation of expressions are also extended, so that binary operations $\oplus$ are homomorphic on integer pairs. Thus, $\sigma(e)$, the evaluation of expression e using state σ, may be either an integer n or an integer pair $(\!|\, v_1 \mid v_2 \,|\!)$.

We extend projection to event pairs ($\lfloor (\!|\, \alpha_1 \mid \alpha_2 \,|\!) \rfloor_i = \alpha_i$) and define projection homomorphically on events. We define projection on traces inductively, as follows.

$$\lfloor \langle\rangle \rfloor_i = \langle\rangle$$

$$\lfloor \langle \alpha_0, \alpha_1, \ldots \rangle \rfloor_i = \begin{cases} \lfloor \langle \alpha_1, \ldots \rangle \rfloor_i & \text{if } \lfloor \alpha_0 \rfloor_i = \mathsf{void} \\ \langle \lfloor \alpha_0 \rfloor_i \rangle \hat{} \lfloor \langle \alpha_1, \ldots \rangle \rfloor_i & \text{otherwise} \end{cases}$$

Finally, we extend projection to joint strategies, so that for any ℓ and t, $\lfloor \omega \rfloor_i(\ell)(t) = \lfloor \omega(\ell)(t) \rfloor_i$.

We indicate IMPI^2 configurations with a bullet ($\bullet$) subscript: $(c, \sigma, t, \omega)_\bullet$. A IMPI^2 configuration represents a pair of IMPI configurations.

The complete operational semantics of IMPI^2 are given in Fig. 5. Note that rules OS^2-ASSIGN, OS^2-SEQ-1, OS^2-SEQ-2, OS^2-IN, OS^2-OUT, OS^2-IF-1, OS^2-IF-2, OS^2-DECLASSIFY, and OS^2-WHILE are similar to their counterparts in the language IMPI. Rules OS^2-IF-1 and OS^2-DECLASSIFY have been modified to be restricted to apply only to integer results of evaluating expression e.

The rule OS^2-PAIR-LIFT evaluates one of the two subcommands of a pair command $(\!|\, c_1 \mid c_2 \,|\!)$. The memory and trace are update to indicate that only one of the two executions represented by the configuration made progress. Thus, $\lfloor \sigma \rfloor_j = \lfloor \sigma' \rfloor_j$ and $\lfloor t \rfloor_j = \lfloor t' \rfloor_j$, where $j \in \{1, 2\}$ is the execution that did not make progress. Note that the small step relation used in the premise is small step relation of language IMPI.

The rule OS^2-PAIR-SKIP applies when the two commands represented by a command pair have both finished executing. This rule removes the command pair.

The rule OS^2-PAIR-IF applies when the evaluation of a conditional expression differs in the two executions represented by the IMPI^2 configuration. This rule introduces a command pair, representing the different branches that may be taken by the two executions. This is the only rule that introduces command pairs.

The rule OS^2-PAIR-DECLASSIFY applies when an expression is declassified in both executions represented by the execution, and the evaluation of the expression is the same in both executions.

B.2.2 Adequacy

The language IMPI^2 is adequate to express the execution of two IMPI programs. We show that the execution of a IMPI^2 program is sound (a step taken by a IMPI^2 program corresponds to one or zero steps taken by its projections) and complete (given two IMPI executions, there is a IMPI^2 execution whose projection agrees with at least one of them). We write $\longrightarrow^=$ for the reflexive closure of $\longrightarrow$.

Lemma 7 (Soundness)**.** *If* $(c, \sigma, t, \omega)_\bullet \longrightarrow (c', \sigma', t', \omega)_\bullet$*, then* $(\lfloor c \rfloor_i, \lfloor \sigma \rfloor_i, \lfloor t \rfloor_i, \lfloor \omega \rfloor_i) \longrightarrow^= (\lfloor c' \rfloor_i, \lfloor \sigma' \rfloor_i, \lfloor t' \rfloor_i, \lfloor \omega \rfloor_i)$ *for*

$i \in \{1, 2\}$.

Proof. By induction on the derivation $(c, \sigma, t, \omega)_\bullet \longrightarrow (c', \sigma', t', \omega)_\bullet$. The interesting cases are the new rules introduced for IMPI^2: OS^2-PAIR-LIFT, OS^2-PAIR-SKIP, OS^2-PAIR-IF, and OS^2-PAIR-DECLASSIFY. For a reduction using OS^2-PAIR-LIFT, clearly one of the two projections takes a step, while the other projection remains unchanged. For OS^2-PAIR-SKIP, both projections remain unchanged. For both OS^2-PAIR-IF, and OS^2-PAIR-DECLASSIFY, both projections take a step. □

If an IMPI^2 configuration is stuck, it is because one of the two projections is stuck.

Lemma 8 (Stuck configurations)**.** *If $(c, \sigma, t, \omega)_\bullet$ is stuck (i.e., cannot be reduced and $c \neq$ skip), then $(\lfloor c \rfloor_i, \lfloor \sigma \rfloor_i, \lfloor t \rfloor_i, \lfloor \omega \rfloor_i)$ is stuck for some $i \in \{1, 2\}$.*

Proof. By structural induction on command c. □

Given two IMPI evaluations, there is an IMPI^2 evaluation that represents the same IMPI evaluation for at least one of the two evaluation.

Lemma 9 (Completeness)**.** *If $(\lfloor c \rfloor_i, \lfloor \sigma \rfloor_i, \lfloor t \rfloor_i, \lfloor \omega \rfloor_i) \longrightarrow^* (c'_i, \sigma'_i, t'_i, \lfloor \omega \rfloor_i)$ for $i \in \{1, 2\}$, then there exists a IMPI^2 configuration $(c', \sigma', t', \omega)_\bullet$ such that $(c, \sigma, t, \omega)_\bullet \longrightarrow^* (c', \sigma', t', \omega)_\bullet$ and $(\lfloor c' \rfloor_i, \lfloor \sigma' \rfloor_i, \lfloor t' \rfloor_i, \lfloor \omega \rfloor_i) = (c'_i, \sigma'_i, t'_i, \lfloor \omega \rfloor_i)$ for some $i \in \{1, 2\}$.*

Proof. Let $\tau_i = (\lfloor c \rfloor_i, \lfloor \sigma \rfloor_i, \lfloor t \rfloor_i, \lfloor \omega \rfloor_i) \ldots (c'_i, \sigma'_i, t'_i, \lfloor \omega \rfloor_i)$ be the sequence of configurations that witnesses $(\lfloor c \rfloor_i, \lfloor \sigma \rfloor_i, \lfloor t \rfloor_i, \lfloor \omega \rfloor_i) \longrightarrow^* (c'_i, \sigma'_i, t'_i, \lfloor \omega \rfloor_i)$.

Let n_i be the length of τ_i. For a sequence IMPI^2 configurations $\tau = (c, \sigma, t, \omega)_\bullet \ldots (c', \sigma', t', \omega)_\bullet$ that witnesses $(c, \sigma, t, \omega)_\bullet \longrightarrow^* (c', \sigma', t', \omega)_\bullet$, let $f_i(\tau)$ be n_i minus the number of reduction steps in τ that reduce the ith projection. Note that $f_i(\tau)$ is non-negative. Consider $g(\tau) = \min(f_1(\tau), f_2(\tau))$. If $g(\tau) = 0$, then τ is a sequence that satisfies the conditions.

Suppose $g(\tau) > 0$. Consider the function

$$h(\tau) = (g(\tau),\ |f_1(\tau) - f_2(\tau)|,\ numPairs(\tau[|\tau| - 1]))$$

where $\tau[|\tau| - 1]$ refers to the last configuration in the sequence τ, and $numPairs((c, \sigma, t, \omega)_\bullet)$ returns the number of pair commands in c. Note that all elements of the triple returned by $h(\tau)$ are non-negative. If we can extend τ by one step to a sequence τ' such that $h(\tau') < h(\tau)$ under lexicographic ordering, then, by repeated applications, eventually we will produce a sequence τ'' such that $g(\tau'') = 0$.

We now show how to extend sequence τ to a sequence τ' such that $h(\tau') < h(\tau)$. By assumption, $g(\tau) > 0$, so neither last configuration of τ_1 or τ_2 is stuck. By Lemma 8, we can extend τ by one more step, producing trace τ'. By Lemma 7, either $f_i(\tau') = f_i(\tau) - 1$ for some $i \in \{1, 2\}$, or $f_i(\tau') = f_i(\tau)$ for all $i \in \{1, 2\}$. If the former, then $h(\tau') < h(\tau)$. If the latter, then the rule OS^2-PAIR-SKIP was used in the reduction, and the last configuration of τ' has one fewer pair commands than the last configuration of τ, and so $h(\tau') < h(\tau)$. □

B.2.3 Type preservation

We extend the type system to IMPI^2 commands and configurations. Typing judgments are now of the form

$$pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$$

where $\ell \in \mathcal{L}$. Intuitively, if IMPI^2 command c is well-typed with typing parameter ℓ, then c represents the two IMPI commands that are indistinguishable from the point of view of any user ℓ' such that $\ell' \sqsubseteq \ell$.

Because our type system is flow-dependent, we need to extend the typing context entities Γ, Δ, and E so that they range over pairs. This allows the expression of different typing contexts for the two IMPI commands represented by a single IMPI^2 command. Thus, Γ ranges over security levels $\mathcal{L}$ and pairs of security levels (written $(\!|\, \ell_1 \mid \ell_2 \,|\!)$. Similarly, Δ and E range over elements of $\mathbf{InputExp}_\perp$ and pairs of elements of $\mathbf{InputExp}_\perp$. We also extend the entity C so that its range is $(\mathbb{Z}_\perp \cup (\mathbb{Z}_\perp \times \mathbb{Z}_\perp)) \times (\mathbb{Z}_\perp \cup (\mathbb{Z}_\perp \times \mathbb{Z}_\perp))$. That is, for any $\ell \in \mathcal{L}$, we have $C(\ell)(i, j)$, where i is either an integer (indicating the number of inputs received on channel ℓ), $\perp$ (indicating an unknown number of inputs received on channel ℓ), or a pair (i_1, i_2), where i_1 indicates inputs received on channel ℓ in the first execution, and i_2 indicates inputs received on channel ℓ in the second execution. Similarly, j describes the outputs sent on channel ℓ in both executions. We define projection operations $\lfloor\cdot\rfloor_1$ and $\lfloor\cdot\rfloor_2$ for all the extended entities. We extend relations $\sqsubseteq$ and $\succeq$ such that

$$\begin{aligned}
\ell \sqsubseteq (\!|\, \ell_1 \mid \ell_2 \,|\!) &\iff \ell \sqsubseteq \ell_1 \text{ and } \ell \sqsubseteq \ell_2 \\
(\!|\, \ell_1 \mid \ell_2 \,|\!) \sqsubseteq \ell &\iff \ell_1 \sqsubseteq \ell \text{ and } \ell_2 \sqsubseteq \ell \\
(\!|\, \ell_1 \mid \ell_2 \,|\!) \sqsubseteq (\!|\, \ell_1' \mid \ell_2' \,|\!) &\iff \ell_1 \sqsubseteq \ell_1' \text{ and } \ell_2 \sqsubseteq \ell_2' \\
v \succeq (\!|\, v_1 \mid v_2 \,|\!) &\iff v \succeq v_1 \text{ and } v \succeq v_2 \\
(\!|\, v_1 \mid v_2 \,|\!) \succeq v &\iff v_1 \succeq v \text{ and } v_2 \succeq v \\
(\!|\, v_1 \mid v_2 \,|\!) \succeq (\!|\, v_1' \mid v_2' \,|\!) &\iff v_1 \succeq v_1' \text{ and } v_2 \succeq v_2'
\end{aligned}$$

Typing rules for IMPI (given in Fig. 3) are made into typing rules for IMPI^2 by adding the typing parameter ℓ to every rule. In addition, we severely restrict when the typing context entities may differ for the two different IMPI commands represented by a single IMPI^2 command. We require for judgment $pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$ that (a) the images of C', Δ', E' and Γ' do not contain any pairs (i.e., they are suitable IMPI entities); and (b) if c does not contain a command pair, then C, Δ, E and Γ do not contain any pairs. We use the predicate $noPairs(\Gamma, C, \Delta, \mathrm{E})$ to indicate that the images of C, Δ, E and Γ do not contain any pairs.

All typing rules for IMPI^2 are presented in Fig. 6. The typing rule for the new pair command, T^2-PAIR, requires that both projections type check using IMPI typing rules, for a program counter level pc' that is at least as restrictive as typing parameter ℓ. Intuitively, this will ensure that any side-effects of a command pair will not be observable at level ℓ or below. Note that the premise of T^2-PAIR uses the typing judgment for IMPI, i.e., without the typing parameter ℓ. This is because

well-formed commands do not have nested command pairs. All typing rules other than T^2-PAIR correspond closely to their IMPI counterpart.

We define a notion of satisfaction for IMPI^2 configurations. Intuitively, an IMPI^2 configuration satisfies context $C, \Delta, \text{E}, \Gamma$ for ℓ if the two IMPI configurations represented by the IMPI^2 configuration are identical at all levels ℓ' such that $\ell' \sqsubseteq \ell$, and each IMPI configuration satisfies the appropriate IMPI context. We also require that no command pair appears as a subcommand of an if or while command.

Definition 7. We say that configuration $(c, \sigma, t, \omega)_\bullet$ satisfies context $C, \Delta, \text{E}, \Gamma$ for ℓ, written $\Gamma, C, \Delta, \text{E} \vDash_\ell (c, \sigma, t, \omega)_\bullet$ if all of the following conditions hold.

1. For all $x \in \textbf{Var}$, if $\sigma(x)$ is a pair value then $\Gamma(x) \not\sqsubseteq \ell$.

2. For all i such that $0 \leq i < |t|$, if $value(t(i))$ is a pair value, then $level(t(i)) \not\sqsubseteq \ell$, where

$$value(\langle\!|\, \alpha_1 \mid \alpha_2 \,|\!\rangle) = \langle\!|\, 0 \mid 0 \,|\!\rangle$$
$$value(in(\ell, v)) = v$$
$$value(out(\ell, v)) = v$$
$$level(\langle\!|\, \mathsf{void} \mid \alpha \,|\!\rangle) = level(\alpha)$$
$$level(\langle\!|\, \alpha \mid \mathsf{void} \,|\!\rangle) = level(\alpha)$$
$$level(in(\ell, v)) = \ell$$
$$level(out(\ell, v)) = \ell$$

3. For all ℓ', t', if $\omega(\ell')(t')$ is a pair value, then $\ell' \not\sqsubseteq \ell$.

4. $\lfloor C \rfloor_i, \lfloor \Delta \rfloor_i, \lfloor \text{E} \rfloor_i, \vDash (\lfloor c \rfloor_i, \lfloor \sigma \rfloor_i, \lfloor t \rfloor_i, \lfloor \omega \rfloor_i)$ for $i \in \{1, 2\}$.

5. No command pair appears as a subcommand of an if or while command of c.

The execution of a IMPI^2 program preserves typings. The following type-preservation theorem is key to showing that well-typed IMPI programs satisfy delimited release.

Theorem 3 (Type preservation)**.** *Let c be an IMPI^2 command such that*

$$pc, \Gamma; C, \Delta, \text{E}, H \vdash_\ell c \triangleright \Gamma'; C', \Delta', \text{E}', H',$$

and let $(c, \sigma, t, \omega)_\bullet$ be an IMPI^2 configuration such that

$$\Gamma, C, \Delta, \text{E} \vDash_\ell (c, \sigma, t, \omega)_\bullet$$

If

$$(c, \sigma, t, \omega)_\bullet \longrightarrow (c', \sigma', t', \omega)_\bullet$$

and $\lfloor t' \rfloor_1$ *and* $\lfloor t' \rfloor_2$ *agree up to* ℓ *on escape hatches* H' *then there exists* C'', Δ'', E'', H'', Γ'', *and* pc'' *such that*

$$pc'', \Gamma''; C'', \Delta'', \mathrm{E}'', H'' \vdash_\ell c' \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$$

and

$$\Gamma'', C'', \Delta'', \mathrm{E}'' \vDash_\ell (c', \sigma', t', \omega)_\bullet.$$

Proof. By induction on the typing judgment $pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$.

□

Before proving that the IMPI type system enforces delimited release, we first prove some useful lemmas about the IMPI^2 type system.

The judgment $\Gamma', C', \Delta', \mathrm{E}' \vDash_\ell (c', \sigma', t, \omega)_\bullet$ ensures that the two traces represented by IMPI^2 trace t are identical to user ℓ. More precisely, all input and output on any channel bounded above by typing parameter ℓ is the same in both executions.

Lemma 10. *If* $\Gamma', C', \Delta', \mathrm{E}' \vDash_\ell (c', \sigma', t, \omega)_\bullet$ *then* $\lfloor t \rfloor_1 \restriction \ell' = \lfloor t \rfloor_2 \restriction \ell'$ *for any* $\ell' \in \mathcal{L}$ *such that* $\ell' \sqsubseteq \ell$.

Proof. By induction on the length of t. The base case, $t = \langle\rangle$, is trivial. Consider $\alpha \hat{} t$, and assume that $\lfloor t \rfloor_1 \restriction \ell' = \lfloor t \rfloor_2 \restriction \ell'$. If $level(\alpha) \neq \ell'$, then $\lfloor \alpha \hat{} t \rfloor_i \restriction \ell' = \lfloor t \rfloor_i \restriction \ell'$ for $i \in \{1, 2\}$, and the result holds. If $level(\alpha) \sqsubseteq \ell'$ then by $\Gamma', C', \Delta', \mathrm{E}' \vDash_\ell (c', \sigma', t, \omega)_\bullet$ we have $value(\alpha)$ is not a pair value. Thus, $\lfloor \alpha \rfloor_1 = \lfloor \alpha \rfloor_2$, and $\lfloor \alpha \hat{} t \rfloor_1 \restriction \ell' = \lfloor \alpha \hat{} t \rfloor_2 \restriction \ell'$ as required. □

If an IMPI command is well-typed in the IMPI type system, then it is well-typed in the IMPI^2 type system.

Lemma 11. *If* c *is an* IMPI *command (i.e., does not contain any command pairs), and*

$$pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'$$

then for all ℓ *we have*

$$pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'.$$

Proof. Every IMPI typing rule is made into an IMPI^2 typing rule by adding the typing parameter ℓ. □

B.2.4 Proof of delimited release

Using the type preservation of IMPI^2, and the lemmas above, we can now show that a well-typed IMPI program satisfies delimited release.

Lemma 12. *If*

$$pc, \Gamma_0; \overline{(0,0)}, \overline{\bot}, \overline{\bot}, \emptyset \vdash c \triangleright \Gamma; C, \Delta, \mathrm{E}, H$$

for some Γ_0 and pc, and for all $(f, \ell) \in H$ there exists $i \in 0..k$ such that $f \equiv f_i$ and $\ell_i \sqsubseteq \ell$, then c satisfies delimited release by escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$.

Proof. Assume $pc, \Gamma_0; \overline{(0,0)}, \overline{\bot}, \overline{\bot}, \emptyset \vdash\ c \triangleright \Gamma; C, \Delta, \mathrm{E}, H$ for some Γ_0 and pc. Let escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$ be fixed, and assume that for all $(f, \ell) \in H$ there exists $i \in 0..k$ such that $f \equiv f_i$ and $\ell_i \sqsubseteq \ell$.

Let $\ell \in \mathcal{L}$. Let ω_1 and ω_2 be joint strategies such that $\omega_1(\ell') = \omega_2(\ell')$ for all $\ell' \sqsubseteq \ell$. Let σ be an initial state, and t_1 and t_2 be traces such that t_1 and t_2 agree up to ℓ on escape hatches $(f_0, \ell_0), \ldots, (f_k, \ell_k)$, and $(c, \sigma, \langle\rangle, \omega_1) \rightsquigarrow t_1$ and $(c, \sigma, \langle\rangle, \omega_2) \rightsquigarrow t_2$.

By Lemma 11, $pc, \Gamma_0; \overline{(0,0)}, \overline{\bot}, \overline{\bot}, \emptyset \vdash_\ell\ c \triangleright \Gamma; C, \Delta, \mathrm{E}, H$. Since for all $(f, \ell') \in H$ there exists $i \in 0..k$ such that $f \equiv f_i$ and $\ell_i \sqsubseteq \ell'$, and so traces t_1 and t_2 agree up to ℓ on escape hatches H.

Let σ be a state that contains no pair values. Let ω be a IMPI^2 joint strategy such that $\lfloor \omega \rfloor_i = \omega_i$, and for any ℓ' and t', if $\ell' \sqsubseteq \ell$ then $\omega(\ell')(t')$ is not a pair value. Note that $\overline{\bot}, \Gamma_0, \overline{(0,0)}, \overline{\bot} \vDash_\ell (c, \sigma, \langle\rangle, \omega)_\bullet$.

Suppose that both t_1 and t_2 are finite traces. Then by Lemma 9 and Lemma 7, there is an IMPI^2 configuration $(c', \sigma', t, \omega)_\bullet$ such that $(c, \sigma, \langle\rangle, \omega)_\bullet \longrightarrow^* (c', \sigma', t, \omega)_\bullet$ and $\lfloor t \rfloor_i = t_i$ and $t_j \succeq \lfloor t \rfloor_j$ for some i and j such that $\{i, j\} = \{1, 2\}$. By repeated applications of Theorem 3, we have $\Gamma', C', \Delta', \mathrm{E}' \vDash_\ell (c', \sigma', t, \omega)_\bullet$ for some C', Δ', E', and Γ'. Thus, by Lemma 10, we have $\lfloor t \rfloor_i \restriction \ell = \lfloor t \rfloor_j \restriction \ell$, and so $t_j \restriction \ell \succeq t_i \restriction \ell$ and thus $t_1 \approx_\ell t_2$ as required.

Let one or both of t_1 or t_2 be an infinite trace. Suppose that it is not the case that either $t_1 \restriction \ell \succeq t_2 \restriction \ell$ or $t_2 \restriction \ell \succeq t_1 \restriction \ell$. Therefore there is some index n such that $(t_1 \restriction \ell)(n) \neq (t_2 \restriction \ell)(n)$. Consider finite traces t_1' and t_2' such that $t_1 \succeq t_1'$ and $t_2 \succeq t_2'$, and $|t_1' \restriction \ell| = |t_2' \restriction \ell| = n + 1$. Note that $(c, \sigma, \langle\rangle, \omega_1) \rightsquigarrow t_1'$ and $(c, \sigma, \langle\rangle, \omega_2) \rightsquigarrow t_2'$. By a similar argument above, we derive that $t_1' \approx_\ell t_2'$. But this implies that $(t_1' \restriction \ell)(n) = (t_2' \restriction \ell)(n)$, a contradiction! Therefore, either $t_1 \restriction \ell \succeq t_2 \restriction \ell$ or $t_2 \restriction \ell \succeq t_1 \restriction \ell$, and so $t_1 \approx_\ell t_2$ as required. □

B.3 Bounded release

The proof of Theorem 2 follows immediately from Lemmas 6 and 12.

OS-ASSIGN

$$\frac{}{(x := e, \sigma, t, \omega) \longrightarrow (\mathsf{skip}, \sigma[x \mapsto \sigma(e)], t, \omega)}$$

OS-SEQ-1

$$\frac{}{(\mathsf{skip}; c, \sigma, t, \omega) \longrightarrow (c, \sigma, t, \omega)}$$

OS-SEQ-2

$$\frac{(c_0, \sigma, t, \omega) \longrightarrow (c_0', \sigma', t', \omega)}{(c_0; c_1, \sigma, t, \omega) \longrightarrow (c_0'; c_1, \sigma', t', \omega)}$$

OS-IN

$$\frac{\omega(\ell)(t \restriction \ell) = v}{(\mathsf{input}\ x\ \mathsf{from}\ \ell, \sigma, t, \omega) \longrightarrow (\mathsf{skip}, \sigma[x \mapsto v], t\hat{}\langle in(\ell, v)\rangle, \omega)}$$

OS-OUT

$$\frac{\sigma(e) = v}{(\mathsf{output}\ e\ \mathsf{to}\ \ell, \sigma, t, \omega) \longrightarrow (\mathsf{skip}, \sigma, t\hat{}\langle out(\ell, v)\rangle, \omega)}$$

OS-IF-1

$$\frac{\sigma(e) \neq 0}{(\mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1, \sigma, t, \omega) \longrightarrow (c_0, \sigma, t, \omega)}$$

OS-IF-2

$$\frac{\sigma(e) = 0}{(\mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1, \sigma, t, \omega) \longrightarrow (c_1, \sigma, t, \omega)}$$

OS-WHILE

$$\frac{}{(\mathsf{while}\ e\ \mathsf{do}\ c, \sigma, t, \omega) \longrightarrow (\mathsf{if}\ e\ \mathsf{then}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}, \sigma, t, \omega)}$$

OS-DECLASSIFY

$$\frac{}{(x := \mathsf{declassify}(e\ \mathsf{to}\ \ell), \sigma, t, \omega) \longrightarrow (\mathsf{skip}, \sigma[x \mapsto \sigma(e)], t, \omega)}$$

Figure 1: Operational semantics

$$\frac{}{t \vDash_{in} n \Downarrow n} \qquad \frac{t \vDash_{in} f_0 \Downarrow v_0 \quad t \vDash_{in} f_1 \Downarrow v_1}{t \vDash_{in} f_0 \oplus f_1 \Downarrow v} \, v = v_0 \oplus v_1 \qquad \frac{t' = t \restriction (\mathbf{Ev}(\ell) \cap \mathbf{Ev}_{in}) \quad t'(i) = in(\ell, v)}{t \vDash_{in} \mathsf{in}_\ell[i] \Downarrow v} \qquad \frac{t' = t \restriction (\mathbf{Ev}(\ell) \cap \mathbf{Ev}_{in}) \quad \neg(0 \leq i < |t'|)}{t \vDash_{in} \mathsf{in}_\ell[i] \Downarrow \bot}$$

$$\frac{}{t \vDash^{\ell}_{out} n \Downarrow n} \qquad \frac{t \vDash^{\ell}_{out} g_0 \Downarrow v_0 \quad t \vDash^{\ell}_{out} g_1 \Downarrow v_1}{t \vDash^{\ell}_{out} g_0 \oplus g_1 \Downarrow v} \, v = v_0 \oplus v_1 \qquad \frac{t' = t \restriction (\mathbf{Ev}(\ell) \cap \mathbf{Ev}_{out}) \quad t'(i) = out(\ell, v)}{t \vDash^{\ell}_{out} \mathsf{out}[i] \Downarrow v} \qquad \frac{t' = t \restriction (\mathbf{Ev}(\ell) \cap \mathbf{Ev}_{out}) \quad \neg(0 \leq i < |t'|)}{t \vDash^{\ell}_{out} \mathsf{out}[i] \Downarrow \bot}$$

Figure 2: Evaluation rules for input and output expressions

T-ASSIGN

$$\frac{pc \sqsubseteq \ell \qquad \Gamma(e) \sqsubseteq \ell \qquad \Gamma' = \Gamma[x \mapsto \ell] \qquad \Delta' = \Delta[x \mapsto \Delta(e)]}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash x := e \triangleright \Gamma'; C, \Delta', \mathrm{E}, H}$$

T-SEQ

$$\frac{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c_0 \triangleright \Gamma'; C', \Delta', \mathrm{E}', H' \qquad pc, \Gamma'; C', \Delta', \mathrm{E}', H' \vdash c_1 \triangleright \Gamma''; C'', \Delta'', \mathrm{E}'', H''}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash c_0; c_1 \triangleright \Gamma''; C'', \Delta'', \mathrm{E}'', H''}$$

T-IN

$$\frac{pc \sqsubseteq \ell \qquad \Gamma' = \Gamma[x \mapsto \ell] \qquad C' = inc_{in}(C, \ell) \qquad \Delta' = recordInput(\Delta, x, C, \ell)}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash \mathsf{input}\ x\ \mathsf{from}\ \ell \triangleright \Gamma'; C', \Delta', \mathrm{E}, H}$$

T-OUT

$$\frac{pc \sqsubseteq \ell \qquad \Gamma(e) \sqsubseteq \ell \qquad C' = inc_{out}(C, \ell) \qquad \mathrm{E}' = recordOutput(\mathrm{E}, C, \ell, \Delta(e))}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash \mathsf{output}\ e\ \mathsf{to}\ \ell \triangleright \Gamma; C', \Delta, \mathrm{E}', H}$$

T-IF

$$\frac{pc \sqsubseteq pc' \qquad \Gamma(e) \sqsubseteq pc' \qquad i = 0, 1 \qquad pc', \Gamma; C, \Delta, \mathrm{E}, H \vdash c_i \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash \mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'}$$

T-WHILE

$$\frac{pc \sqsubseteq pc' \qquad \Gamma(e) \sqsubseteq pc' \qquad pc', \Gamma; C, \Delta, \mathrm{E}, H \vdash c \triangleright \Gamma; C, \Delta, \mathrm{E}, H}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash \mathsf{while}\ e\ \mathsf{do}\ c \triangleright \Gamma; C, \Delta, \mathrm{E}, H}$$

T-DECLASSIFY

$$\frac{pc \sqsubseteq \ell' \qquad \ell \sqsubseteq \ell' \qquad \Gamma' = \Gamma[x \mapsto \ell'] \qquad \Delta' = \Delta[x \mapsto \Delta(e)] \qquad H' = H \cup \{(\Delta(e), \ell)\}}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash x := \mathsf{declassify}(e\ \mathsf{to}\ \ell) \triangleright \Gamma'; C, \Delta', \mathrm{E}, H'}$$

T-SKIP

$$\frac{}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash \mathsf{skip} \triangleright \Gamma; C, \Delta, \mathrm{E}, H}$$

T-SUB

$$\frac{\begin{array}{c}\Gamma_0 \sqsubseteq \Gamma_1 \qquad \Gamma_1' \sqsubseteq \Gamma_0' \qquad pc_0 \sqsubseteq pc_1 \\ C_0 \succeq C_1 \quad C_1' \succeq C_0' \qquad \Delta_0 \succeq \Delta_1 \quad \Delta_1' \succeq \Delta_0' \\ \mathrm{E}_0 \succeq \mathrm{E}_1 \quad \mathrm{E}_1' \succeq \mathrm{E}_0' \qquad H_0 \subseteq H_1 \quad H_1' \subseteq H_0' \\ pc_1, \Gamma_1; C_1, \Delta_1, \mathrm{E}_1, H_1 \vdash c \triangleright \Gamma_1'; C_1', \Delta_1', \mathrm{E}_1', H_1'\end{array}}{pc_0, \Gamma_0; C_0, \Delta_0, \mathrm{E}_0, H_0 \vdash c \triangleright \Gamma_0'; C_0', \Delta_0', \mathrm{E}_0', H_0'}$$

$$recordInput(\Delta, x, C, \ell) = \begin{cases} \Delta[x \mapsto \bot] & \text{if } C(\ell) = (\bot, j) \\ \Delta[x \mapsto \mathsf{in}_\ell[i]] & \text{if } C(\ell) = (i, j), i \neq \bot \end{cases}$$

$$recordOutput(\mathrm{E}, C, \ell, f) = \begin{cases} \mathrm{E} & \text{if } C(\ell) = (i, \bot) \\ \mathrm{E}[(\ell, j) \mapsto f] & \text{if } C(\ell) = (i, j), j \neq \bot \end{cases}$$

$$inc_{in}(C, \ell) = \begin{cases} C & \text{if } C(\ell) = (\bot, j) \\ C[\ell \mapsto (i+1, j)] & \text{if } C(\ell) = (i, j), i \neq \bot \end{cases}$$

$$inc_{out}(C, \ell) = \begin{cases} C & \text{if } C(\ell) = (i, \bot) \\ C[\ell \mapsto (i, j+1)] & \text{if } C(\ell) = (i, j), j \neq \bot \end{cases}$$

Figure 3: Typing rules

$$
\begin{aligned}
\mathit{substOutExp}(\mathrm{E}, \ell, n) &= n \\
\mathit{substOutExp}(\mathrm{E}, \ell, g_0 \oplus g_1) &= \mathit{substOutExp}(\mathrm{E}, \ell, g_0) \oplus \\
&\quad\ \mathit{substOutExp}(\mathrm{E}, \ell, g_1) \\
\mathit{substOutExp}(\mathrm{E}, \ell, \mathsf{out}[i]) &= \mathrm{E}(\ell, i)
\end{aligned}
$$

Figure 4: $\mathit{substOutExp}(\mathrm{E}, \ell, g)$

OS^2-ASSIGN

$$\frac{}{(x := e, \sigma, t, \omega)_\bullet \longrightarrow (\mathsf{skip}, \sigma[x \mapsto \sigma(e)], t, \omega)_\bullet}$$

OS^2-SEQ-1

$$\frac{}{(\mathsf{skip}; c, \sigma, t, \omega)_\bullet \longrightarrow (c, \sigma, t, \omega)_\bullet}$$

OS^2-SEQ-2

$$\frac{(c_0, \sigma, t, \omega)_\bullet \longrightarrow (c_0', \sigma', t', \omega)_\bullet}{(c_0; c_1, \sigma, t, \omega)_\bullet \longrightarrow (c_0'; c_1, \sigma', t', \omega)_\bullet}$$

OS^2-IN

$$\frac{\omega(\ell)(t \restriction \ell) = v}{(\mathsf{input}\ x\ \mathsf{from}\ \ell, \sigma, t, \omega)_\bullet \longrightarrow (\mathsf{skip}, \sigma[x \mapsto v], t\hat{}\langle in(\ell, v)\rangle, \omega)_\bullet}$$

OS^2-OUT

$$\frac{\sigma(e) = v}{(\mathsf{output}\ e\ \mathsf{to}\ \ell, \sigma, t, \omega)_\bullet \longrightarrow (\mathsf{skip}, \sigma, t\hat{}\langle out(\ell, v)\rangle, \omega)_\bullet}$$

OS^2-IF-1

$$\frac{\sigma(e) \neq 0 \quad \sigma(e) \in \mathbb{Z}}{(\mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1, \sigma, t, \omega)_\bullet \longrightarrow (c_0, \sigma, t, \omega)_\bullet}$$

OS^2-IF-2

$$\frac{\sigma(e) = 0}{(\mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1, \sigma, t, \omega)_\bullet \longrightarrow (c_1, \sigma, t, \omega)_\bullet}$$

OS^2-PAIR-SKIP

$$\frac{}{(\langle\!|\, \mathsf{skip} \mid \mathsf{skip} \,|\!\rangle, \sigma, t, \omega)_\bullet \longrightarrow (\mathsf{skip}, \sigma, t, \omega)_\bullet}$$

OS^2-DECLASSIFY

$$\frac{\sigma(e) \neq \langle\!|\, v \mid v \,|\!\rangle}{(x := \mathsf{declassify}(e\ \mathsf{to}\ \ell), \sigma, t, \omega)_\bullet \longrightarrow (\mathsf{skip}, \sigma[x \mapsto \sigma(e)], t, \omega)_\bullet}$$

OS^2-WHILE

$$\frac{}{(\mathsf{while}\ e\ \mathsf{do}\ c, \sigma, t, \omega)_\bullet \longrightarrow (\mathsf{if}\ e\ \mathsf{then}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}, \sigma, t, \omega)_\bullet}$$

OS^2-PAIR-LIFT

$$\frac{\begin{array}{c}\{i,j\} = \{1,2\} \quad (c_i, \lfloor\sigma\rfloor_i, \lfloor t\rfloor_i, \lfloor\omega\rfloor_i) \longrightarrow (c_i', \sigma_i', t_i', \lfloor\omega\rfloor_i) \quad c_j' = c_j \\ \sigma' = \lambda x.\ \begin{cases} \langle\!|\, \sigma_1'(x) \mid \sigma_2'(x) \,|\!\rangle & \text{if } \lfloor\sigma\rfloor_i(x) \neq \sigma_i'(x) \\ \sigma(x) & \text{otherwise} \end{cases} \\ t' = \begin{cases} t & \text{if } \lfloor t\rfloor_i = t_i' \\ t\hat{}\langle\langle\!|\, \alpha_0 \mid \alpha_1 \,|\!\rangle\rangle & \text{if } \lfloor t\rfloor_i \neq t_i' \end{cases} \\ \alpha_i = t_i'(|t_i'| - 1) \quad \alpha_j = \mathsf{void} \end{array}}{(\langle\!|\, c_1 \mid c_2 \,|\!\rangle, \sigma, t, \omega)_\bullet \longrightarrow (\langle\!|\, c_1' \mid c_2' \,|\!\rangle, \sigma', t', \omega)_\bullet}$$

OS^2-PAIR-IF

$$\frac{\sigma(e) = \langle\!|\, v_1 \mid v_2 \,|\!\rangle \qquad c_i' = \begin{cases} c_0 & \text{if } v_i \neq 0 \\ c_1 & \text{if } v_i = 0 \end{cases}}{(\mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1, \sigma, t, \omega)_\bullet \longrightarrow (\langle\!|\, c_1' \mid c_2' \,|\!\rangle, \sigma, t, \omega)_\bullet}$$

OS^2-PAIR-DECLASSIFY

$$\frac{\sigma(e) = \langle\!|\, v \mid v \,|\!\rangle}{(x := \mathsf{declassify}(e\ \mathsf{to}\ \ell), \sigma, t, \omega)_\bullet \longrightarrow (\mathsf{skip}, \sigma[x \mapsto v], t, \omega)_\bullet}$$

Figure 5: Operational semantics of IMPI^2

T^2-ASSIGN

$$\frac{pc \sqsubseteq \ell' \quad \Gamma(e) \sqsubseteq \ell' \quad noPairs(\Gamma, C, \Delta, \mathrm{E}) \quad \Gamma' = \Gamma[x \mapsto \ell'] \quad \Delta' = \Delta[x \mapsto \Delta(e)]}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell x := e \triangleright \Gamma'; C, \Delta', \mathrm{E}, H}$$

T^2-SKIP

$$\frac{noPairs(\Gamma, C, \Delta, \mathrm{E})}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell \mathsf{skip} \triangleright \Gamma; C, \Delta, \mathrm{E}, H}$$

T^2-SEQ

$$\frac{\begin{array}{c} pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c_0 \triangleright \Gamma'; C', \Delta', \mathrm{E}', H' \\ pc, \Gamma'; C', \Delta', \mathrm{E}', H' \vdash_\ell c_1 \triangleright \Gamma''; C'', \Delta'', \mathrm{E}'', H'' \\ noPairs(\Gamma', C', \Delta', \mathrm{E}') \quad noPairs(\Gamma'', C'', \Delta'', \mathrm{E}'') \\ \text{if } c_0 \text{ does not contain any command pairs then } noPairs(\Gamma, C, \Delta, \mathrm{E}) \end{array}}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c_0; c_1 \triangleright \Gamma''; C'', \Delta'', \mathrm{E}'', H''}$$

T^2-IN

$$\frac{pc \sqsubseteq \ell' \quad \Gamma' = \Gamma[x \mapsto \ell'] \quad noPairs(\Gamma, C, \Delta, \mathrm{E}) \quad C' = inc_{in}(C, \ell') \quad \Delta' = recordInput(\Delta, x, C, \ell')}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell \mathsf{input}\ x\ \mathsf{from}\ \ell' \triangleright \Gamma'; C', \Delta', \mathrm{E}, H}$$

T^2-OUT

$$\frac{pc \sqsubseteq \ell' \quad \Gamma(e) \sqsubseteq \ell' \quad noPairs(\Gamma, C, \Delta, \mathrm{E}) \quad C' = inc_{out}(C, \ell') \quad \mathrm{E}' = recordOutput(\mathrm{E}, C, \ell', \Delta(e))}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell \mathsf{output}\ e\ \mathsf{to}\ \ell' \triangleright \Gamma; C', \Delta, \mathrm{E}', H}$$

T^2-IF

$$\frac{\begin{array}{c} pc \sqsubseteq pc' \quad \Gamma(e) \sqsubseteq pc' \quad i = 0, 1 \quad noPairs(\Gamma', C', \Delta', \mathrm{E}') \\ pc', \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c_i \triangleright \Gamma'; C', \Delta', \mathrm{E}', H' \\ \text{if } c_i \text{ does not contain command pairs then } noPairs(\Gamma, C, \Delta, \mathrm{E}) \end{array}}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell \mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'}$$

T^2-WHILE

$$\frac{pc \sqsubseteq pc' \quad \Gamma(e) \sqsubseteq pc' \quad noPairs(\Gamma, C, \Delta, \mathrm{E}) \quad pc', \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell c \triangleright \Gamma; C, \Delta, \mathrm{E}, H}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell \mathsf{while}\ e\ \mathsf{do}\ c \triangleright \Gamma; C, \Delta, \mathrm{E}, H}$$

T^2-DECLASSIFY

$$\frac{pc \sqsubseteq \ell'' \quad \ell' \sqsubseteq \ell'' \quad \Gamma' = \Gamma[x \mapsto \ell''] \quad noPairs(\Gamma, C, \Delta, \mathrm{E}) \quad \Delta' = \Delta[x \mapsto \Delta(e)] \quad H' = H \cup \{(\Delta(e), \ell')\}}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell x := \mathsf{declassify}(e\ \mathsf{to}\ \ell') \triangleright \Gamma'; C, \Delta', \mathrm{E}, H'}$$

T^2-SUB

$$\frac{\begin{array}{c} \Gamma_0 \sqsubseteq \Gamma_1 \quad \Gamma_1' \sqsubseteq \Gamma_0' \quad pc_0 \sqsubseteq pc_1 \quad noPairs(\Gamma_0', C_0', \Delta_0', \mathrm{E}_0') \\ C_0 \succeq C_1 \quad C_1' \succeq C_0' \quad \Delta_0 \succeq \Delta_1 \quad \Delta_1' \succeq \Delta_0' \\ \mathrm{E}_0 \succeq \mathrm{E}_1 \quad \mathrm{E}_1' \succeq \mathrm{E}_0' \quad H_0 \subseteq H_1 \quad H_1' \subseteq H_0' \\ pc_1, \Gamma_1; C_1, \Delta_1, \mathrm{E}_1, H_1 \vdash_\ell c \triangleright \Gamma_1'; C_1', \Delta_1', \mathrm{E}_1', H_1' \\ \text{if } c \text{ does not contain any command pairs then } noPairs(\Gamma_0, C_0, \Delta_0, \mathrm{E}_0) \end{array}}{pc_0, \Gamma_0; C_0, \Delta_0, \mathrm{E}_0, H_0 \vdash_\ell c \triangleright \Gamma_0'; C_0', \Delta_0', \mathrm{E}_0', H_0'}$$

T^2-PAIR

$$\frac{pc \sqsubseteq pc' \quad pc' \not\sqsubseteq \ell \quad noPairs(\Gamma', C', \Delta', \mathrm{E}') \quad i = 1, 2 \quad pc', \lfloor\Gamma\rfloor_i; \lfloor C\rfloor_i, \lfloor\Delta\rfloor_i, \lfloor\mathrm{E}\rfloor_i, H \vdash c_i \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'}{pc, \Gamma; C, \Delta, \mathrm{E}, H \vdash_\ell (\!| c_1 \mid c_2 |\!) \triangleright \Gamma'; C', \Delta', \mathrm{E}', H'}$$

Figure 6: Typing rules